4050 Final Project--——— Image Recognition of American Sign Language

Group 2: Shijie An, Xiaoying Lin, Xinchang Liu, Zhen Xu, Yaxuan Yang

1. Introduction

Sign languages

Sign languages (also known as signed languages) are languages that use manual communication to convey meaning. This can include simultaneously employing hand gestures, movement, orientation of the fingers, arms or body, and facial expressions to convey a speaker's ideas.

Linguists consider both spoken and signed communication to be types of natural language, meaning that both emerged through an abstract, protracted aging process and evolved over time without meticulous planning. Sign language should not be confused with body language, a type of nonverbal communication.

How do We Apply the Dataset in Education?

Online quiz section in sign language online learning platform, to improve the interaction of self-learning process Students make the sign language in front of the the computer, camera capture the image Image uploaded to the models Models identify whether the student has made the correct gesture

2. Dataset Description

In this dataset, there are 10 classes, which each of them represents the gesture from 1 to 10. Each picture is 100x100 pixels, and there are 218 students participated to give number gestures. There are totally 2062 pictures.

3. EDA

In [ ]:

```
## import all the required packages.
import os
import numpy as np
from os import listdir
from imageio import imread
from keras.utils import to_categorical
from sklearn.model_selection import train_test_split
from keras.utils.image_utils import img_to_array

import PIL
import matplotlib.pyplot as plt
```

In [ ]:
```
# Settings
num_classes = 10
test_size = 0.2
```

This function is used to read the picture from the `data_path` and convert the picture to black and white

In [ ]:
```
def get_img(data_path):
  ## Getting image array from path:
  img = PIL.Image.open(data_path)
  img = img.convert("L")
  img = img_to_array(img)
  img = np.resize(img, (100, 100, 1))
  return img
```

Get dataset from picture and then split to train and test set

In [ ]:
```
dataset_path = "/content/drive/MyDrive/HUDK_4050_Final/Dataset"

## Getting all data from data path
labels = sorted(listdir(dataset_path)) ## in order to read the files from the sorted li
X = []
Y = []
for i, label in enumerate(labels):
  data_path = dataset_path + "/" + label

  for data in listdir(data_path):
    ## create dataset
    img = get_img(data_path + "/" + data)
    X.append(img) ## X is the source file for all pictures.
    Y.append(i)   ## Y is the number represented for all the picutres.
## transfer X, and Y
X = 1 - np.array(X).astype("float32") /255
Y = np.array(Y).astype("float32")
Y = to_categorical(Y, num_classes)
## split out the dataset
X, X_test, Y, Y_test = train_test_split(X, Y, test_size=test_size, random_state = 42)
print(X.shape)
print(X_test.shape)
print(Y.shape)
print(Y_test.shape)
```

```
(1649, 100, 100, 1)
(413, 100, 100, 1)
(1649, 10)
(413, 10)
```

4. Model and results

4.1 Fully Connected Structure

In [ ]:
```
#Import all needed libraries
import os
import numpy as np
from os import listdir
from imageio import imread
from keras.utils import to_categorical
from sklearn.model_selection import train_test_split
from keras.utils.image_utils import img_to_array

import PIL
import matplotlib.pyplot as plt

from tensorflow import keras
import numpy as np
import pandas as pd
import sklearn as sk
import time
from keras.datasets import mnist
from keras.models import Sequential, load_model
from keras.layers import Dense, Dropout, Flatten
from keras import optimizers
from keras import backend as K
from keras import regularizers
from keras import initializers
import keras as ks
from matplotlib import pyplot as plt
```

In [ ]:
```
#Connected to my google drive
from google.colab import drive
drive.mount('/content/drive')
```

```
Mounted at /content/drive
```

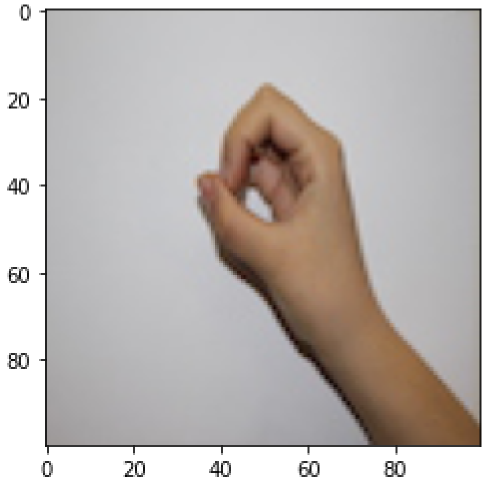
In [ ]:
```
#Settings
num_classes = 10
test_size = 0.2
```

In [ ]:
```
#Read image and convert to 3D array
def get_img(data_path):
  ## Getting image array from path:
  img = PIL.Image.open(data_path)
  img = img.convert("L")
  img = img_to_array(img)
  img = np.resize(img, (100, 100))
#   img = np.load(data_path)
  return img
```

In [ ]:
```
#Get dataset form pictures and split totrain and test sets
from matplotlib import image
from matplotlib import pyplot
#Load image as pixel array
```

```
image = image.imread('/content/drive/MyDrive/4050_Final_Dataset/0/IMG_1118.JPG')
#Summarize shape of the pixel array
print(image.dtype)
print(image.shape)
#Display the array of pixels as an image
pyplot.imshow(image)
pyplot.show()
```

```
uint8
(100, 100, 3)
```



In [ ]:

```
dataset_path = "/content/drive/MyDrive/4050_Final_Dataset"

## Getting all data from data path
labels = sorted(listdir(dataset_path))
print(labels)
X = []
Y = []
for i, label in enumerate(labels):
  data_path = dataset_path + "/" + label

  for data in listdir(data_path):
    img = get_img(data_path + "/" + data)
    X.append(img)
    Y.append(i)
## create dataset
X = 1 - np.array(X).astype("float32") /255
# X = np.array(X).astype("float32")
Y = np.array(Y).astype("float32")
Y = to_categorical(Y, num_classes)

X, X_test, Y, Y_test = train_test_split(X, Y, test_size=test_size, random_state = 42)
print(X.shape)
print(X_test.shape)
print(Y.shape)
print(Y_test.shape)
```

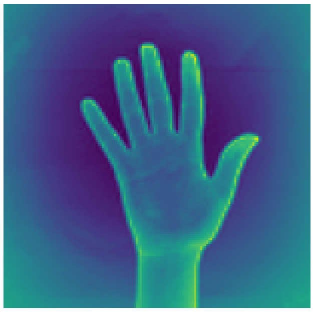
```
['0', '1', '2', '3', '4', '5', '6', '7', '8', '9']
(1649, 100, 100)
(413, 100, 100)
(1649, 10)
(413, 10)
```

In [ ]:

```
img_size = 64

plt.subplot(1 , 2 , 1)
```

```
plt.imshow(X[0])
plt.axis("off")
```

Out[ ]: (-0.5, 99.5, 99.5, -0.5)



In [ ]:
```
from keras.utils.np_utils import to_categorical
## unroll the height and width and thickness into one big vector
x_train = X.reshape(1649, 10000)
x_test = X_test.reshape(413, 10000)
x_train = x_train.astype("float32")
x_test = x_test.astype("float32")

## normalize pixel values from 0 to 255
# data is already normalized
# x_train /= 255
# x_test /= 255


# y_train = to_categorical(Y, 10)
# Y_test = to_categorical(Y_test, 10)
y_train = Y
y_test = Y_test

print(x_train.shape)
print(y_train.shape)
print(x_test.shape)
print(y_test.shape)
```

```
(1649, 10000)
(1649, 10)
(413, 10000)
(413, 10)
```

Part A: The general idea is that I kept using neural network with one hidden layer and one dropout, with momentum stochastic gradient descent, different activation functions, batch size and epoch. I used activation function "sigmoid" at first, with batch size = 50. As the accuracy is pretty lower, at around 0.10, I changed parameters many times. For example, I changed batch size by increasing it gradually. As it still not working, I decided to use other activation function such as softmax. Softmax is used for normalizing the outputs, since it can convert them from weighted sum values into probabilities that sum to one. The value in the output of the softmax function will be interpreted as the prbability of membership for each class. And I also decreased the value of batch size. As a result, the accuracy increased to 0.40~0.50. The accuracy enhanced to the largest (0.7191) with batch size = 36 and activation function = sigmoid.

In [ ]:
```
model = Sequential()
model.add(Dense(10, activation='sigmoid', input_dim =x_train.shape[1]))
#One hidden layer + one dropout
```

```
model.add(Dropout(0.1))
model.add(Dense(5, activation='sigmoid'))
model.add(Dense(10, activation='sigmoid'))

model.summary()

#Momentum Stochastic Gradient Descent
sgd = keras.optimizers.SGD(lr=0.01, decay=1e-6, momentum=0.9, nesterov=True)

#Compile the model
model.compile(loss='categorical_crossentropy',
              optimizer=sgd, metrics=['accuracy'])
```

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 dense (Dense)               (None, 10)                100010

 dropout (Dropout)           (None, 10)                0

 dense_1 (Dense)             (None, 5)                 55

 dense_2 (Dense)             (None, 10)                60

=================================================================
Total params: 100,125
Trainable params: 100,125
Non-trainable params: 0
_________________________________________________________________
/usr/local/lib/python3.8/dist-packages/keras/optimizers/optimizer_v2/gradient_descent.p
y:108: UserWarning: The `lr` argument is deprecated, use `learning_rate` instead.
  super(SGD, self).__init__(name, **kwargs)
```

In [ ]:

```
model.fit(x_train, y_train,batch_size=32,
               epochs=100, verbose=2)
```

```
Epoch 1/100
52/52 - 1s - loss: 2.3173 - accuracy: 0.1079 - 758ms/epoch - 15ms/step
Epoch 2/100
52/52 - 0s - loss: 2.2974 - accuracy: 0.1249 - 179ms/epoch - 3ms/step
Epoch 3/100
52/52 - 0s - loss: 2.2865 - accuracy: 0.1395 - 159ms/epoch - 3ms/step
Epoch 4/100
52/52 - 0s - loss: 2.2759 - accuracy: 0.1589 - 156ms/epoch - 3ms/step
Epoch 5/100
52/52 - 0s - loss: 2.2624 - accuracy: 0.1953 - 166ms/epoch - 3ms/step
Epoch 6/100
52/52 - 0s - loss: 2.2444 - accuracy: 0.2116 - 169ms/epoch - 3ms/step
Epoch 7/100
52/52 - 0s - loss: 2.2238 - accuracy: 0.2256 - 151ms/epoch - 3ms/step
Epoch 8/100
52/52 - 0s - loss: 2.1953 - accuracy: 0.2438 - 164ms/epoch - 3ms/step
Epoch 9/100
52/52 - 0s - loss: 2.1607 - accuracy: 0.2771 - 164ms/epoch - 3ms/step
Epoch 10/100
52/52 - 0s - loss: 2.1240 - accuracy: 0.2577 - 158ms/epoch - 3ms/step
Epoch 11/100
52/52 - 0s - loss: 2.0832 - accuracy: 0.2735 - 153ms/epoch - 3ms/step
Epoch 12/100
52/52 - 0s - loss: 2.0333 - accuracy: 0.2996 - 154ms/epoch - 3ms/step
Epoch 13/100
52/52 - 0s - loss: 1.9854 - accuracy: 0.3184 - 154ms/epoch - 3ms/step
Epoch 14/100
```

```
52/52 - 0s - loss: 1.9273 - accuracy: 0.3354 - 153ms/epoch - 3ms/step
Epoch 15/100
52/52 - 0s - loss: 1.8778 - accuracy: 0.3517 - 160ms/epoch - 3ms/step
Epoch 16/100
52/52 - 0s - loss: 1.8299 - accuracy: 0.3493 - 156ms/epoch - 3ms/step
Epoch 17/100
52/52 - 0s - loss: 1.7756 - accuracy: 0.3954 - 162ms/epoch - 3ms/step
Epoch 18/100
52/52 - 0s - loss: 1.7421 - accuracy: 0.3863 - 166ms/epoch - 3ms/step
Epoch 19/100
52/52 - 0s - loss: 1.6905 - accuracy: 0.4148 - 158ms/epoch - 3ms/step
Epoch 20/100
52/52 - 0s - loss: 1.6585 - accuracy: 0.4196 - 157ms/epoch - 3ms/step
Epoch 21/100
52/52 - 0s - loss: 1.6349 - accuracy: 0.4196 - 165ms/epoch - 3ms/step
Epoch 22/100
52/52 - 0s - loss: 1.6012 - accuracy: 0.4263 - 163ms/epoch - 3ms/step
Epoch 23/100
52/52 - 0s - loss: 1.5675 - accuracy: 0.4524 - 171ms/epoch - 3ms/step
Epoch 24/100
52/52 - 0s - loss: 1.5340 - accuracy: 0.4700 - 147ms/epoch - 3ms/step
Epoch 25/100
52/52 - 0s - loss: 1.5117 - accuracy: 0.4633 - 154ms/epoch - 3ms/step
Epoch 26/100
52/52 - 0s - loss: 1.4811 - accuracy: 0.4839 - 154ms/epoch - 3ms/step
Epoch 27/100
52/52 - 0s - loss: 1.4638 - accuracy: 0.4779 - 168ms/epoch - 3ms/step
Epoch 28/100
52/52 - 0s - loss: 1.4406 - accuracy: 0.4773 - 158ms/epoch - 3ms/step
Epoch 29/100
52/52 - 0s - loss: 1.4189 - accuracy: 0.5070 - 160ms/epoch - 3ms/step
Epoch 30/100
52/52 - 0s - loss: 1.3991 - accuracy: 0.5112 - 160ms/epoch - 3ms/step
Epoch 31/100
52/52 - 0s - loss: 1.3761 - accuracy: 0.5215 - 157ms/epoch - 3ms/step
Epoch 32/100
52/52 - 0s - loss: 1.3520 - accuracy: 0.5294 - 153ms/epoch - 3ms/step
Epoch 33/100
52/52 - 0s - loss: 1.3268 - accuracy: 0.5434 - 162ms/epoch - 3ms/step
Epoch 34/100
52/52 - 0s - loss: 1.3239 - accuracy: 0.5549 - 158ms/epoch - 3ms/step
Epoch 35/100
52/52 - 0s - loss: 1.3045 - accuracy: 0.5555 - 152ms/epoch - 3ms/step
Epoch 36/100
52/52 - 0s - loss: 1.3024 - accuracy: 0.5488 - 157ms/epoch - 3ms/step
Epoch 37/100
52/52 - 0s - loss: 1.2761 - accuracy: 0.5531 - 151ms/epoch - 3ms/step
Epoch 38/100
52/52 - 0s - loss: 1.2440 - accuracy: 0.5858 - 163ms/epoch - 3ms/step
Epoch 39/100
52/52 - 0s - loss: 1.2452 - accuracy: 0.5585 - 149ms/epoch - 3ms/step
Epoch 40/100
52/52 - 0s - loss: 1.2239 - accuracy: 0.5797 - 165ms/epoch - 3ms/step
Epoch 41/100
52/52 - 0s - loss: 1.2108 - accuracy: 0.5797 - 156ms/epoch - 3ms/step
Epoch 42/100
52/52 - 0s - loss: 1.2081 - accuracy: 0.5858 - 175ms/epoch - 3ms/step
Epoch 43/100
52/52 - 0s - loss: 1.1770 - accuracy: 0.6064 - 150ms/epoch - 3ms/step
Epoch 44/100
52/52 - 0s - loss: 1.1684 - accuracy: 0.6113 - 152ms/epoch - 3ms/step
Epoch 45/100
52/52 - 0s - loss: 1.1617 - accuracy: 0.6173 - 153ms/epoch - 3ms/step
Epoch 46/100
52/52 - 0s - loss: 1.1252 - accuracy: 0.6173 - 164ms/epoch - 3ms/step
```

```
Epoch 47/100
52/52 - 0s - loss: 1.1066 - accuracy: 0.6374 - 153ms/epoch - 3ms/step
Epoch 48/100
52/52 - 0s - loss: 1.0940 - accuracy: 0.6446 - 156ms/epoch - 3ms/step
Epoch 49/100
52/52 - 0s - loss: 1.0733 - accuracy: 0.6531 - 152ms/epoch - 3ms/step
Epoch 50/100
52/52 - 0s - loss: 1.0784 - accuracy: 0.6489 - 154ms/epoch - 3ms/step
Epoch 51/100
52/52 - 0s - loss: 1.0640 - accuracy: 0.6458 - 153ms/epoch - 3ms/step
Epoch 52/100
52/52 - 0s - loss: 1.0644 - accuracy: 0.6604 - 156ms/epoch - 3ms/step
Epoch 53/100
52/52 - 0s - loss: 1.0206 - accuracy: 0.7047 - 168ms/epoch - 3ms/step
Epoch 54/100
52/52 - 0s - loss: 1.0138 - accuracy: 0.6810 - 150ms/epoch - 3ms/step
Epoch 55/100
52/52 - 0s - loss: 0.9997 - accuracy: 0.6828 - 156ms/epoch - 3ms/step
Epoch 56/100
52/52 - 0s - loss: 0.9942 - accuracy: 0.6895 - 154ms/epoch - 3ms/step
Epoch 57/100
52/52 - 0s - loss: 0.9785 - accuracy: 0.6931 - 153ms/epoch - 3ms/step
Epoch 58/100
52/52 - 0s - loss: 0.9469 - accuracy: 0.7132 - 155ms/epoch - 3ms/step
Epoch 59/100
52/52 - 0s - loss: 0.9393 - accuracy: 0.7095 - 159ms/epoch - 3ms/step
Epoch 60/100
52/52 - 0s - loss: 0.9294 - accuracy: 0.7216 - 155ms/epoch - 3ms/step
Epoch 61/100
52/52 - 0s - loss: 0.9265 - accuracy: 0.7144 - 155ms/epoch - 3ms/step
Epoch 62/100
52/52 - 0s - loss: 0.9013 - accuracy: 0.7271 - 151ms/epoch - 3ms/step
Epoch 63/100
52/52 - 0s - loss: 0.8980 - accuracy: 0.7314 - 151ms/epoch - 3ms/step
Epoch 64/100
52/52 - 0s - loss: 0.8922 - accuracy: 0.7338 - 156ms/epoch - 3ms/step
Epoch 65/100
52/52 - 0s - loss: 0.8475 - accuracy: 0.7526 - 170ms/epoch - 3ms/step
Epoch 66/100
52/52 - 0s - loss: 0.8636 - accuracy: 0.7508 - 157ms/epoch - 3ms/step
Epoch 67/100
52/52 - 0s - loss: 0.8575 - accuracy: 0.7489 - 157ms/epoch - 3ms/step
Epoch 68/100
52/52 - 0s - loss: 0.8267 - accuracy: 0.7641 - 153ms/epoch - 3ms/step
Epoch 69/100
52/52 - 0s - loss: 0.8414 - accuracy: 0.7532 - 150ms/epoch - 3ms/step
Epoch 70/100
52/52 - 0s - loss: 0.8070 - accuracy: 0.7629 - 154ms/epoch - 3ms/step
Epoch 71/100
52/52 - 0s - loss: 0.8034 - accuracy: 0.7635 - 168ms/epoch - 3ms/step
Epoch 72/100
52/52 - 0s - loss: 0.7898 - accuracy: 0.7659 - 153ms/epoch - 3ms/step
Epoch 73/100
52/52 - 0s - loss: 0.7920 - accuracy: 0.7774 - 157ms/epoch - 3ms/step
Epoch 74/100
52/52 - 0s - loss: 0.7694 - accuracy: 0.7799 - 155ms/epoch - 3ms/step
Epoch 75/100
52/52 - 0s - loss: 0.7616 - accuracy: 0.7768 - 154ms/epoch - 3ms/step
Epoch 76/100
52/52 - 0s - loss: 0.7780 - accuracy: 0.7744 - 155ms/epoch - 3ms/step
Epoch 77/100
52/52 - 0s - loss: 0.7603 - accuracy: 0.7714 - 161ms/epoch - 3ms/step
Epoch 78/100
52/52 - 0s - loss: 0.7457 - accuracy: 0.7908 - 166ms/epoch - 3ms/step
Epoch 79/100
```

```
52/52 - 0s - loss: 0.7247 - accuracy: 0.8047 - 155ms/epoch - 3ms/step
Epoch 80/100
52/52 - 0s - loss: 0.7224 - accuracy: 0.7944 - 159ms/epoch - 3ms/step
Epoch 81/100
52/52 - 0s - loss: 0.7362 - accuracy: 0.7702 - 156ms/epoch - 3ms/step
Epoch 82/100
52/52 - 0s - loss: 0.7066 - accuracy: 0.7962 - 160ms/epoch - 3ms/step
Epoch 83/100
52/52 - 0s - loss: 0.6860 - accuracy: 0.8059 - 158ms/epoch - 3ms/step
Epoch 84/100
52/52 - 0s - loss: 0.6883 - accuracy: 0.8084 - 167ms/epoch - 3ms/step
Epoch 85/100
52/52 - 0s - loss: 0.6924 - accuracy: 0.8041 - 156ms/epoch - 3ms/step
Epoch 86/100
52/52 - 0s - loss: 0.6919 - accuracy: 0.8035 - 158ms/epoch - 3ms/step
Epoch 87/100
52/52 - 0s - loss: 0.6621 - accuracy: 0.8035 - 161ms/epoch - 3ms/step
Epoch 88/100
52/52 - 0s - loss: 0.6529 - accuracy: 0.8187 - 154ms/epoch - 3ms/step
Epoch 89/100
52/52 - 0s - loss: 0.6694 - accuracy: 0.8102 - 158ms/epoch - 3ms/step
Epoch 90/100
52/52 - 0s - loss: 0.6456 - accuracy: 0.8150 - 169ms/epoch - 3ms/step
Epoch 91/100
52/52 - 0s - loss: 0.6237 - accuracy: 0.8381 - 154ms/epoch - 3ms/step
Epoch 92/100
52/52 - 0s - loss: 0.6367 - accuracy: 0.8181 - 155ms/epoch - 3ms/step
Epoch 93/100
52/52 - 0s - loss: 0.6130 - accuracy: 0.8326 - 155ms/epoch - 3ms/step
Epoch 94/100
52/52 - 0s - loss: 0.6305 - accuracy: 0.8241 - 155ms/epoch - 3ms/step
Epoch 95/100
52/52 - 0s - loss: 0.6210 - accuracy: 0.8223 - 158ms/epoch - 3ms/step
Epoch 96/100
52/52 - 0s - loss: 0.6135 - accuracy: 0.8229 - 155ms/epoch - 3ms/step
Epoch 97/100
52/52 - 0s - loss: 0.6035 - accuracy: 0.8308 - 168ms/epoch - 3ms/step
Epoch 98/100
52/52 - 0s - loss: 0.6152 - accuracy: 0.8193 - 160ms/epoch - 3ms/step
Epoch 99/100
52/52 - 0s - loss: 0.5721 - accuracy: 0.8532 - 158ms/epoch - 3ms/step
Epoch 100/100
52/52 - 0s - loss: 0.5979 - accuracy: 0.8266 - 156ms/epoch - 3ms/step
```

Out[ ]: <keras.callbacks.History at 0x7fbe8e392520>

In [ ]:

```
#Evaluate with test data
model.evaluate(x_test, y_test, batch_size=32)
```

```
13/13 [==============================] - 0s 2ms/step - loss: 0.9987 - accuracy: 0.6780
```

Out[ ]: [0.9987353086471558, 0.6779661178588867]

Part B: In this part, I changed the activation function to relu, and used Nesterov momentum stochastic gradient descent, dropouts, L2 regularization and random Gaussian weight initialization with 1/sqrt(n) standard deviation. Specifically, for layers, I created from 1 layers and kept adding to 3 layers. For epoch, the smallest I used is 5, and the largest I used is 100. By changing layers, dropouts, batch size, and epoch, I got the highest accuracy at around 0.20.

In [ ]:

```
#create a model structure, fit the model with train data, evaluate with test data
#Neural Network with three layers
model = Sequential()
model.add(Dense(32, activation='relu', input_dim =x_train.shape[1], kernel_regularizer=
```

```
                kernel_initializer=initializers.RandomNormal(mean=0,stddev=0.036)))
#hidden layer + dropout
model.add(Dropout(0.1))
model.add(Dense(32, activation='relu'))
model.add(Dense(10, activation='relu'))
model.summary()

#Momentum Stochastic Gradient Descent
sgd = keras.optimizers.SGD(lr=0.01, decay=1e-6, momentum=0.9, nesterov=True)

#Compile the model
model.compile(loss='categorical_crossentropy',
              optimizer=sgd, metrics=['accuracy'])
```

```
Model: "sequential_1"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 dense_3 (Dense)             (None, 32)                320032

 dropout_1 (Dropout)         (None, 32)                0

 dense_4 (Dense)             (None, 32)                1056

 dense_5 (Dense)             (None, 10)                330

=================================================================
Total params: 321,418
Trainable params: 321,418
Non-trainable params: 0
_________________________________________________________________
```

In [ ]:
```
model.fit(x_train, y_train,batch_size=32,
                epochs = 100, verbose=2)
```

```
Epoch 1/100
52/52 - 1s - loss: 9.0494 - accuracy: 0.1055 - 738ms/epoch - 14ms/step
Epoch 2/100
52/52 - 0s - loss: 8.0545 - accuracy: 0.1061 - 218ms/epoch - 4ms/step
Epoch 3/100
52/52 - 0s - loss: 7.4265 - accuracy: 0.1061 - 210ms/epoch - 4ms/step
Epoch 4/100
52/52 - 0s - loss: 6.9296 - accuracy: 0.1061 - 225ms/epoch - 4ms/step
Epoch 5/100
52/52 - 0s - loss: 6.5296 - accuracy: 0.1031 - 205ms/epoch - 4ms/step
Epoch 6/100
52/52 - 0s - loss: 6.2059 - accuracy: 0.0995 - 225ms/epoch - 4ms/step
Epoch 7/100
52/52 - 0s - loss: 5.9445 - accuracy: 0.1037 - 212ms/epoch - 4ms/step
Epoch 8/100
52/52 - 0s - loss: 5.7332 - accuracy: 0.1061 - 208ms/epoch - 4ms/step
Epoch 9/100
52/52 - 0s - loss: 5.5616 - accuracy: 0.1079 - 211ms/epoch - 4ms/step
Epoch 10/100
52/52 - 0s - loss: 5.4216 - accuracy: 0.0988 - 214ms/epoch - 4ms/step
Epoch 11/100
52/52 - 0s - loss: 5.3092 - accuracy: 0.1092 - 222ms/epoch - 4ms/step
Epoch 12/100
52/52 - 0s - loss: 5.2183 - accuracy: 0.1164 - 208ms/epoch - 4ms/step
Epoch 13/100
52/52 - 0s - loss: 5.1390 - accuracy: 0.1383 - 219ms/epoch - 4ms/step
Epoch 14/100
52/52 - 0s - loss: 4.6605 - accuracy: 0.1322 - 210ms/epoch - 4ms/step
Epoch 15/100
```

```
52/52 - 0s - loss: 4.0226 - accuracy: 0.1031 - 241ms/epoch - 5ms/step
Epoch 16/100
52/52 - 0s - loss: 3.9240 - accuracy: 0.1164 - 212ms/epoch - 4ms/step
Epoch 17/100
52/52 - 0s - loss: 3.8475 - accuracy: 0.0988 - 210ms/epoch - 4ms/step
Epoch 18/100
52/52 - 0s - loss: 3.7888 - accuracy: 0.0946 - 230ms/epoch - 4ms/step
Epoch 19/100
52/52 - 0s - loss: 3.7397 - accuracy: 0.1025 - 212ms/epoch - 4ms/step
Epoch 20/100
52/52 - 0s - loss: 3.6982 - accuracy: 0.1043 - 224ms/epoch - 4ms/step
Epoch 21/100
52/52 - 0s - loss: 3.6671 - accuracy: 0.0970 - 207ms/epoch - 4ms/step
Epoch 22/100
52/52 - 0s - loss: 3.6385 - accuracy: 0.1098 - 213ms/epoch - 4ms/step
Epoch 23/100
52/52 - 0s - loss: 3.6182 - accuracy: 0.1025 - 207ms/epoch - 4ms/step
Epoch 24/100
52/52 - 0s - loss: 3.5975 - accuracy: 0.1128 - 211ms/epoch - 4ms/step
Epoch 25/100
52/52 - 0s - loss: 3.5801 - accuracy: 0.1140 - 222ms/epoch - 4ms/step
Epoch 26/100
52/52 - 0s - loss: 3.5598 - accuracy: 0.1267 - 224ms/epoch - 4ms/step
Epoch 27/100
52/52 - 0s - loss: 3.5327 - accuracy: 0.1304 - 211ms/epoch - 4ms/step
Epoch 28/100
52/52 - 0s - loss: 2.8226 - accuracy: 0.1383 - 211ms/epoch - 4ms/step
Epoch 29/100
52/52 - 0s - loss: 2.5897 - accuracy: 0.1013 - 230ms/epoch - 4ms/step
Epoch 30/100
52/52 - 0s - loss: 2.5479 - accuracy: 0.1007 - 203ms/epoch - 4ms/step
Epoch 31/100
52/52 - 0s - loss: 2.4744 - accuracy: 0.0946 - 218ms/epoch - 4ms/step
Epoch 32/100
52/52 - 0s - loss: 2.4416 - accuracy: 0.1037 - 206ms/epoch - 4ms/step
Epoch 33/100
52/52 - 0s - loss: 2.4163 - accuracy: 0.1001 - 211ms/epoch - 4ms/step
Epoch 34/100
52/52 - 0s - loss: 2.3950 - accuracy: 0.1055 - 219ms/epoch - 4ms/step
Epoch 35/100
52/52 - 0s - loss: 2.3763 - accuracy: 0.1007 - 209ms/epoch - 4ms/step
Epoch 36/100
52/52 - 0s - loss: 2.3633 - accuracy: 0.0934 - 214ms/epoch - 4ms/step
Epoch 37/100
52/52 - 0s - loss: 2.3523 - accuracy: 0.0849 - 205ms/epoch - 4ms/step
Epoch 38/100
52/52 - 0s - loss: 2.3431 - accuracy: 0.1067 - 215ms/epoch - 4ms/step
Epoch 39/100
52/52 - 0s - loss: 2.3353 - accuracy: 0.1067 - 224ms/epoch - 4ms/step
Epoch 40/100
52/52 - 0s - loss: 2.3301 - accuracy: 0.0885 - 212ms/epoch - 4ms/step
Epoch 41/100
52/52 - 0s - loss: 2.3249 - accuracy: 0.0946 - 212ms/epoch - 4ms/step
Epoch 42/100
52/52 - 0s - loss: 2.3212 - accuracy: 0.0958 - 219ms/epoch - 4ms/step
Epoch 43/100
52/52 - 0s - loss: 2.3164 - accuracy: 0.0964 - 222ms/epoch - 4ms/step
Epoch 44/100
52/52 - 0s - loss: 2.3165 - accuracy: 0.1031 - 213ms/epoch - 4ms/step
Epoch 45/100
52/52 - 0s - loss: 2.3120 - accuracy: 0.1031 - 213ms/epoch - 4ms/step
Epoch 46/100
52/52 - 0s - loss: 2.3119 - accuracy: 0.0928 - 204ms/epoch - 4ms/step
Epoch 47/100
52/52 - 0s - loss: 2.3114 - accuracy: 0.0946 - 222ms/epoch - 4ms/step
```

```
Epoch 48/100
52/52 - 0s - loss: 2.3083 - accuracy: 0.1007 - 219ms/epoch - 4ms/step
Epoch 49/100
52/52 - 0s - loss: 2.3082 - accuracy: 0.1061 - 218ms/epoch - 4ms/step
Epoch 50/100
52/52 - 0s - loss: 2.3068 - accuracy: 0.0982 - 212ms/epoch - 4ms/step
Epoch 51/100
52/52 - 0s - loss: 2.3070 - accuracy: 0.0946 - 230ms/epoch - 4ms/step
Epoch 52/100
52/52 - 0s - loss: 2.3063 - accuracy: 0.1061 - 213ms/epoch - 4ms/step
Epoch 53/100
52/52 - 0s - loss: 2.3054 - accuracy: 0.1055 - 216ms/epoch - 4ms/step
Epoch 54/100
52/52 - 0s - loss: 2.3047 - accuracy: 0.0988 - 212ms/epoch - 4ms/step
Epoch 55/100
52/52 - 0s - loss: 2.3045 - accuracy: 0.0964 - 214ms/epoch - 4ms/step
Epoch 56/100
52/52 - 0s - loss: 2.3045 - accuracy: 0.0982 - 204ms/epoch - 4ms/step
Epoch 57/100
52/52 - 0s - loss: 2.3035 - accuracy: 0.0928 - 230ms/epoch - 4ms/step
Epoch 58/100
52/52 - 0s - loss: 2.3058 - accuracy: 0.0976 - 211ms/epoch - 4ms/step
Epoch 59/100
52/52 - 0s - loss: 2.3043 - accuracy: 0.1019 - 216ms/epoch - 4ms/step
Epoch 60/100
52/52 - 0s - loss: 2.3038 - accuracy: 0.1007 - 207ms/epoch - 4ms/step
Epoch 61/100
52/52 - 0s - loss: 2.3022 - accuracy: 0.1043 - 216ms/epoch - 4ms/step
Epoch 62/100
52/52 - 0s - loss: 2.2992 - accuracy: 0.1237 - 214ms/epoch - 4ms/step
Epoch 63/100
52/52 - 0s - loss: 2.2844 - accuracy: 0.1486 - 207ms/epoch - 4ms/step
Epoch 64/100
52/52 - 0s - loss: 2.2488 - accuracy: 0.1649 - 220ms/epoch - 4ms/step
Epoch 65/100
52/52 - 0s - loss: 2.2353 - accuracy: 0.1534 - 211ms/epoch - 4ms/step
Epoch 66/100
52/52 - 0s - loss: 2.2412 - accuracy: 0.1516 - 207ms/epoch - 4ms/step
Epoch 67/100
52/52 - 0s - loss: 2.2036 - accuracy: 0.1722 - 227ms/epoch - 4ms/step
Epoch 68/100
52/52 - 0s - loss: 2.3390 - accuracy: 0.1001 - 211ms/epoch - 4ms/step
Epoch 69/100
52/52 - 0s - loss: 2.3214 - accuracy: 0.0849 - 220ms/epoch - 4ms/step
Epoch 70/100
52/52 - 0s - loss: 2.3181 - accuracy: 0.1001 - 208ms/epoch - 4ms/step
Epoch 71/100
52/52 - 0s - loss: 2.3144 - accuracy: 0.1019 - 217ms/epoch - 4ms/step
Epoch 72/100
52/52 - 0s - loss: 2.3127 - accuracy: 0.1061 - 210ms/epoch - 4ms/step
Epoch 73/100
52/52 - 0s - loss: 2.3111 - accuracy: 0.0995 - 217ms/epoch - 4ms/step
Epoch 74/100
52/52 - 0s - loss: 2.3095 - accuracy: 0.1061 - 208ms/epoch - 4ms/step
Epoch 75/100
52/52 - 0s - loss: 2.3093 - accuracy: 0.0964 - 217ms/epoch - 4ms/step
Epoch 76/100
52/52 - 0s - loss: 2.3081 - accuracy: 0.0898 - 221ms/epoch - 4ms/step
Epoch 77/100
52/52 - 0s - loss: 2.3074 - accuracy: 0.0988 - 214ms/epoch - 4ms/step
Epoch 78/100
52/52 - 0s - loss: 2.3068 - accuracy: 0.0982 - 215ms/epoch - 4ms/step
Epoch 79/100
52/52 - 0s - loss: 2.3058 - accuracy: 0.0922 - 224ms/epoch - 4ms/step
Epoch 80/100
```

```
52/52 - 0s - loss: 2.3059 - accuracy: 0.1061 - 207ms/epoch - 4ms/step
Epoch 81/100
52/52 - 0s - loss: 2.3051 - accuracy: 0.0885 - 221ms/epoch - 4ms/step
Epoch 82/100
52/52 - 0s - loss: 2.3049 - accuracy: 0.0976 - 222ms/epoch - 4ms/step
Epoch 83/100
52/52 - 0s - loss: 2.3053 - accuracy: 0.0940 - 219ms/epoch - 4ms/step
Epoch 84/100
52/52 - 0s - loss: 2.3055 - accuracy: 0.0916 - 212ms/epoch - 4ms/step
Epoch 85/100
52/52 - 0s - loss: 2.3050 - accuracy: 0.1061 - 219ms/epoch - 4ms/step
Epoch 86/100
52/52 - 0s - loss: 2.3043 - accuracy: 0.0940 - 209ms/epoch - 4ms/step
Epoch 87/100
52/52 - 0s - loss: 2.3048 - accuracy: 0.1061 - 223ms/epoch - 4ms/step
Epoch 88/100
52/52 - 0s - loss: 2.3045 - accuracy: 0.0879 - 216ms/epoch - 4ms/step
Epoch 89/100
52/52 - 0s - loss: 2.3045 - accuracy: 0.0922 - 215ms/epoch - 4ms/step
Epoch 90/100
52/52 - 0s - loss: 2.3045 - accuracy: 0.1049 - 219ms/epoch - 4ms/step
Epoch 91/100
52/52 - 0s - loss: 2.3051 - accuracy: 0.0964 - 212ms/epoch - 4ms/step
Epoch 92/100
52/52 - 0s - loss: 2.3069 - accuracy: 0.0946 - 211ms/epoch - 4ms/step
Epoch 93/100
52/52 - 0s - loss: 2.3060 - accuracy: 0.1061 - 214ms/epoch - 4ms/step
Epoch 94/100
52/52 - 0s - loss: 2.3055 - accuracy: 0.0946 - 210ms/epoch - 4ms/step
Epoch 95/100
52/52 - 0s - loss: 2.3053 - accuracy: 0.0970 - 215ms/epoch - 4ms/step
Epoch 96/100
52/52 - 0s - loss: 2.3050 - accuracy: 0.1061 - 216ms/epoch - 4ms/step
Epoch 97/100
52/52 - 0s - loss: 2.3049 - accuracy: 0.0976 - 207ms/epoch - 4ms/step
Epoch 98/100
52/52 - 0s - loss: 2.3048 - accuracy: 0.0952 - 210ms/epoch - 4ms/step
Epoch 99/100
52/52 - 0s - loss: 2.3048 - accuracy: 0.1001 - 218ms/epoch - 4ms/step
Epoch 100/100
52/52 - 0s - loss: 2.3045 - accuracy: 0.0946 - 214ms/epoch - 4ms/step
```

Out[ ]:
```
<keras.callbacks.History at 0x7fbe8e21d9d0>
```

In [ ]:
```python
#Evaluate with test data
model.evaluate(x_test, y_test, batch_size=32)
```

```
13/13 [==============================] - 0s 4ms/step - loss: 2.3092 - accuracy: 0.0775
```

Out[ ]:
```
[2.3091976642608643, 0.07748184353113174]
```

Part C: In this part, I used tuner to help me generate the best network model. However, the highest accuracy is at around 0.30, which is smaller than the accuracy in Part A.

In [ ]:
```
!pip install keras-tuner --upgrade
```

```
Looking in indexes: https://pypi.org/simple, https://us-python.pkg.dev/colab-wheels/publ
ic/simple/
Requirement already satisfied: keras-tuner in /usr/local/lib/python3.8/dist-packages (1.
1.3)
Requirement already satisfied: ipython in /usr/local/lib/python3.8/dist-packages (from k
eras-tuner) (7.9.0)
Requirement already satisfied: requests in /usr/local/lib/python3.8/dist-packages (from
keras-tuner) (2.23.0)
Requirement already satisfied: tensorboard in /usr/local/lib/python3.8/dist-packages (fr
```

```
om keras-tuner) (2.9.1)
Requirement already satisfied: numpy in /usr/local/lib/python3.8/dist-packages (from ker
as-tuner) (1.21.6)
Requirement already satisfied: packaging in /usr/local/lib/python3.8/dist-packages (from
keras-tuner) (21.3)
Requirement already satisfied: kt-legacy in /usr/local/lib/python3.8/dist-packages (from
keras-tuner) (1.0.4)
Requirement already satisfied: prompt-toolkit<2.1.0,>=2.0.0 in /usr/local/lib/python3.8/
dist-packages (from ipython->keras-tuner) (2.0.10)
Requirement already satisfied: traitlets>=4.2 in /usr/local/lib/python3.8/dist-packages
(from ipython->keras-tuner) (5.6.0)
Requirement already satisfied: jedi>=0.10 in /usr/local/lib/python3.8/dist-packages (fro
m ipython->keras-tuner) (0.18.2)
Requirement already satisfied: setuptools>=18.5 in /usr/local/lib/python3.8/dist-package
s (from ipython->keras-tuner) (57.4.0)
Requirement already satisfied: pickleshare in /usr/local/lib/python3.8/dist-packages (fr
om ipython->keras-tuner) (0.7.5)
Requirement already satisfied: backcall in /usr/local/lib/python3.8/dist-packages (from
ipython->keras-tuner) (0.2.0)
Requirement already satisfied: pygments in /usr/local/lib/python3.8/dist-packages (from
ipython->keras-tuner) (2.6.1)
Requirement already satisfied: decorator in /usr/local/lib/python3.8/dist-packages (from
ipython->keras-tuner) (4.4.2)
Requirement already satisfied: pexpect in /usr/local/lib/python3.8/dist-packages (from i
python->keras-tuner) (4.8.0)
Requirement already satisfied: parso<0.9.0,>=0.8.0 in /usr/local/lib/python3.8/dist-pack
ages (from jedi>=0.10->ipython->keras-tuner) (0.8.3)
Requirement already satisfied: wcwidth in /usr/local/lib/python3.8/dist-packages (from p
rompt-toolkit<2.1.0,>=2.0.0->ipython->keras-tuner) (0.2.5)
Requirement already satisfied: six>=1.9.0 in /usr/local/lib/python3.8/dist-packages (fro
m prompt-toolkit<2.1.0,>=2.0.0->ipython->keras-tuner) (1.15.0)
Requirement already satisfied: pyparsing!=3.0.5,>=2.0.2 in /usr/local/lib/python3.8/dist
-packages (from packaging->keras-tuner) (3.0.9)
Requirement already satisfied: ptyprocess>=0.5 in /usr/local/lib/python3.8/dist-packages
(from pexpect->ipython->keras-tuner) (0.7.0)
Requirement already satisfied: chardet<4,>=3.0.2 in /usr/local/lib/python3.8/dist-packag
es (from requests->keras-tuner) (3.0.4)
Requirement already satisfied: urllib3!=1.25.0,!=1.25.1,<1.26,>=1.21.1 in /usr/local/li
b/python3.8/dist-packages (from requests->keras-tuner) (1.24.3)
Requirement already satisfied: certifi>=2017.4.17 in /usr/local/lib/python3.8/dist-packa
ges (from requests->keras-tuner) (2022.9.24)
Requirement already satisfied: idna<3,>=2.5 in /usr/local/lib/python3.8/dist-packages (f
rom requests->keras-tuner) (2.10)
Requirement already satisfied: protobuf<3.20,>=3.9.2 in /usr/local/lib/python3.8/dist-pa
ckages (from tensorboard->keras-tuner) (3.19.6)
Requirement already satisfied: wheel>=0.26 in /usr/local/lib/python3.8/dist-packages (fr
om tensorboard->keras-tuner) (0.38.4)
Requirement already satisfied: google-auth-oauthlib<0.5,>=0.4.1 in /usr/local/lib/python
3.8/dist-packages (from tensorboard->keras-tuner) (0.4.6)
Requirement already satisfied: google-auth<3,>=1.6.3 in /usr/local/lib/python3.8/dist-pa
ckages (from tensorboard->keras-tuner) (2.15.0)
Requirement already satisfied: werkzeug>=1.0.1 in /usr/local/lib/python3.8/dist-packages
(from tensorboard->keras-tuner) (1.0.1)
Requirement already satisfied: tensorboard-data-server<0.7.0,>=0.6.0 in /usr/local/lib/p
ython3.8/dist-packages (from tensorboard->keras-tuner) (0.6.1)
Requirement already satisfied: tensorboard-plugin-wit>=1.6.0 in /usr/local/lib/python3.
8/dist-packages (from tensorboard->keras-tuner) (1.8.1)
Requirement already satisfied: markdown>=2.6.8 in /usr/local/lib/python3.8/dist-packages
(from tensorboard->keras-tuner) (3.4.1)
Requirement already satisfied: absl-py>=0.4 in /usr/local/lib/python3.8/dist-packages (f
rom tensorboard->keras-tuner) (1.3.0)
Requirement already satisfied: grpcio>=1.24.3 in /usr/local/lib/python3.8/dist-packages
(from tensorboard->keras-tuner) (1.51.1)
Requirement already satisfied: pyasn1-modules>=0.2.1 in /usr/local/lib/python3.8/dist-pa
ckages (from google-auth<3,>=1.6.3->tensorboard->keras-tuner) (0.2.8)
```

```
Requirement already satisfied: cachetools<6.0,>=2.0.0 in /usr/local/lib/python3.8/dist-p
ackages (from google-auth<3,>=1.6.3->tensorboard->keras-tuner) (5.2.0)
Requirement already satisfied: rsa<5,>=3.1.4 in /usr/local/lib/python3.8/dist-packages
(from google-auth<3,>=1.6.3->tensorboard->keras-tuner) (4.9)
Requirement already satisfied: requests-oauthlib>=0.7.0 in /usr/local/lib/python3.8/dist
-packages (from google-auth-oauthlib<0.5,>=0.4.1->tensorboard->keras-tuner) (1.3.1)
Requirement already satisfied: importlib-metadata>=4.4 in /usr/local/lib/python3.8/dist-
packages (from markdown>=2.6.8->tensorboard->keras-tuner) (4.13.0)
Requirement already satisfied: zipp>=0.5 in /usr/local/lib/python3.8/dist-packages (from
importlib-metadata>=4.4->markdown>=2.6.8->tensorboard->keras-tuner) (3.11.0)
Requirement already satisfied: pyasn1<0.5.0,>=0.4.6 in /usr/local/lib/python3.8/dist-pac
kages (from pyasn1-modules>=0.2.1->google-auth<3,>=1.6.3->tensorboard->keras-tuner) (0.
4.8)
Requirement already satisfied: oauthlib>=3.0.0 in /usr/local/lib/python3.8/dist-packages
(from requests-oauthlib>=0.7.0->google-auth-oauthlib<0.5,>=0.4.1->tensorboard->keras-tun
er) (3.2.2)
```

In [ ]:

```
import keras_tuner
from tensorflow import keras
from tensorflow.keras import layers
```

In [ ]:

```
# Neural Network with two hidden layers
def build_model(hp):
    # hyperparam we want to tune: number of neurons, lr, activation func
    activation_func = hp.Choice("activation", ["sigmoid","relu"])
    learning_rate = hp.Float("lr", min_value=1e-3, max_value=1e-1, sampling="log")
    neuron_num = hp.Int("neuron-1", min_value=32, max_value=128, step=32)

    # create a model with two hidden layers
    model = Sequential()
    model.add((Dense(32, activation='sigmoid', input_dim =x_train.shape[1])))
    model.add(Dense(units=neuron_num,
                    activation=activation_func))
    model.add(Dense(units=neuron_num,
                    activation=activation_func))
    model.add(Dense(10, activation=activation_func))

    sgd = keras.optimizers.SGD(lr=learning_rate, decay=1e-6, momentum=0.9, nesterov=Tru

    #Compile the model
    model.compile(loss='categorical_crossentropy',
                  optimizer=sgd, metrics=['accuracy'])
    return model

build_model(keras_tuner.HyperParameters())
tuner = keras_tuner.RandomSearch(
    build_model,
    objective='val_loss',
    max_trials=10)
```

In [ ]:

```
tuner.search(x_train, y_train, epochs=5, validation_data=(x_test, y_test))
best_model = tuner.get_best_models()[0]
```

```
WARNING:tensorflow:Detecting that an object or model or tf.train.Checkpoint is being del
eted with unrestored values. See the following logs for the specific values in question.
To silence these warnings, use `status.expect_partial()`. See https://www.tensorflow.or
g/api_docs/python/tf/train/Checkpoint#restorefor details about the status object returne
d by the restore function.
WARNING:tensorflow:Value in checkpoint could not be found in the restored object: (roo
t).layer_with_weights-3.kernel
WARNING:tensorflow:Value in checkpoint could not be found in the restored object: (roo
```

```
t).layer_with_weights-3.bias
WARNING:tensorflow:Value in checkpoint could not be found in the restored object: (roo
t).optimizer.iter
WARNING:tensorflow:Value in checkpoint could not be found in the restored object: (roo
t).optimizer.decay
WARNING:tensorflow:Value in checkpoint could not be found in the restored object: (roo
t).optimizer.learning_rate
WARNING:tensorflow:Value in checkpoint could not be found in the restored object: (roo
t).optimizer.momentum
WARNING:tensorflow:Value in checkpoint could not be found in the restored object: (roo
t).optimizer's state 'momentum' for (root).layer_with_weights-3.kernel
WARNING:tensorflow:Value in checkpoint could not be found in the restored object: (roo
t).optimizer's state 'momentum' for (root).layer_with_weights-3.bias
```

In [ ]:

```
best_model.evaluate(x_test, y_test)
```

```
13/13 [==============================] - 0s 3ms/step - loss: 1.9399 - accuracy: 0.3002
```

Out[ ]:

```
[1.939896821975708, 0.3002421259880066]
```

In [ ]:

```
tuner.results_summary()
```

```
Results summary
Results in ./untitled_project
Showing 10 best trials
<keras_tuner.engine.objective.Objective object at 0x7fcbfa2bc5b0>
Trial summary
Hyperparameters:
activation: sigmoid
lr: 0.0633606054822938
neuron-1: 96
Score: 1.939896821975708
Trial summary
Hyperparameters:
activation: sigmoid
lr: 0.0031531212959390142
neuron-1: 64
Score: 2.3012585639953613
Trial summary
Hyperparameters:
activation: sigmoid
lr: 0.00198317724026896
neuron-1: 64
Score: 2.305431842803955
Trial summary
Hyperparameters:
activation: sigmoid
lr: 0.007116103500829736
neuron-1: 96
Score: 2.3058016300201416
Trial summary
Hyperparameters:
activation: sigmoid
lr: 0.0012066632140502118
neuron-1: 64
Score: 2.307534694671631
Trial summary
Hyperparameters:
activation: relu
lr: 0.012341315610031297
neuron-1: 32
Score: 3.4939591884613037
Trial summary
Hyperparameters:
activation: relu
```

```
lr: 0.0062658328965630354
neuron-1: 64
Score: 4.135528564453125
Trial summary
Hyperparameters:
activation: relu
lr: 0.0028925868757120306
neuron-1: 64
Score: 5.072015762329102
Trial summary
Hyperparameters:
activation: relu
lr: 0.018977192017024146
neuron-1: 64
Score: 5.930873870849609
Trial summary
Hyperparameters:
activation: relu
lr: 0.07006893747238556
neuron-1: 32
Score: 7.032344341278076
```

In [ ]:
```
best_model.summary()
```

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 dense (Dense)               (None, 32)                320032

 dense_1 (Dense)             (None, 96)                3168

 dense_2 (Dense)             (None, 96)                9312

 dense_3 (Dense)             (None, 10)                970

=================================================================
Total params: 333,482
Trainable params: 333,482
Non-trainable params: 0
_________________________________________________________________
```

In [ ]:
```
tuner.search_space_summary()
```

```
Search space summary
Default search space size: 3
activation (Choice)
{'default': 'sigmoid', 'conditions': [], 'values': ['sigmoid', 'relu'], 'ordered': Fals
e}
lr (Float)
{'default': 0.001, 'conditions': [], 'min_value': 0.001, 'max_value': 0.1, 'step': None,
'sampling': 'log'}
neuron-1 (Int)
{'default': None, 'conditions': [], 'min_value': 32, 'max_value': 128, 'step': 32, 'samp
ling': None}
```

4.2 Fully Connected Structure--Using Keras Tuner

run first

In [ ]:
```
from tensorflow import keras
from tensorflow import keras as ks
import numpy as np
import pandas as pd
import sklearn as sk
import time
from keras.datasets import mnist
from keras.models import Sequential, load_model
from keras.layers import Dense, Dropout, Flatten, BatchNormalization
from keras import optimizers

from keras import backend as K
from keras import regularizers
from keras import initializers
from tensorflow.keras import layers
from matplotlib import pyplot as plt

from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
from tensorflow.keras.utils import to_categorical
```

Preprocess the data into right format

In [ ]:
```
## unroll the height and width and thickness into one big vector
x_train = X.reshape(1649, 10000)
x_test = X_test.reshape(413, 10000)
x_train = x_train.astype("float32")
x_test = x_test.astype("float32")

## normalize pixel values from 0 to 255
x_train /= 255
x_test /= 255

y_train = Y
y_test = Y_test
```

set up learning rate from various Dacay rate

In [ ]:
```
import tensorflow
## exponential Decay
initial_learning_rate = 0.1
exponential = keras.optimizers.schedules.ExponentialDecay(
    initial_learning_rate,
    decay_steps=100000,
    decay_rate=0.96,
    staircase=True)

# Piecewise Constant Decay ===> learning rate nan
step = tensorflow.Variable(0, trainable=False)
boundaries = [100000, 110000]
values = [1.0, 0.5, 0.1]
piecewise = keras.optimizers.schedules.PiecewiseConstantDecay(
```

```
    boundaries, values)
# Later, whenever we perform an optimization step, we pass in the step.
# learning_rate = piecewise(step)

# Polynomial Decay ====> best performance
starter_learning_rate = 0.1
end_learning_rate = 0.01
decay_steps = 10000
polynomial = keras.optimizers.schedules.PolynomialDecay(
    starter_learning_rate,
    decay_steps,
    end_learning_rate,
    power=0.5)
```

try tuner

The best result is 0.55

In [ ]:
```
# set hyper-parameters
batch_size = 128
num_classes = 10
epochs = 5
```

The tuner part is to use `keras.tuner` to find out the best number of neruals and what activation function is to use for each layer. Also, this can test how many layer should we get from the nerual network.

I tried to set up the first layer as Dense layer, and set up with the min neural value as 16, the max as 4096, and to step up with 16. The activation choice are `relu` , `sigmoid` , `tanh` , and try to use l1/l2 as kernel_regularizer.

There is a for loop in the middle of the code, which I tried to set up a 2-10 layer for network to find out the best layer number I can have.

In [ ]:
```
import math
try:
  import keras_tuner
except:
  !pip install keras-tuner --upgrade
finally:
  import keras_tuner

def build_model(hp):
    model = keras.Sequential()
    # model.add(layers.Flatten())
    # Tune the number of layers.
    # 原来是min = 16, max = 4096, step = 16
    model.add(Dense(units=hp.Int("1", min_value=16, max_value=4096, step=16),
                    activation=hp.Choice("activation", ["relu", "sigmoid", "tanh"]),
                    input_shape = (10000, ),
                kernel_regularizer = regularizers.l2(0.001),
                kernel_initializer=initializers.RandomNormal(mean=0, stddev = 1/math.sq
    for i in range(hp.Int("num_layers", 2, 10)):
        model.add(
            layers.Dense(
                # Tune number of units separately.
```

```
                units=hp.Int(f"units_{i}", min_value=16, max_value=4096, step=16),
                activation=hp.Choice("activation", ["relu", "sigmoid", "tanh"]),
            )
        )
        #
        # model.add(layers.BatchNormalization())
    if hp.Boolean("dropout"):
        model.add(layers.Dropout(rate=0.2))
    #
    model.add(layers.BatchNormalization())
    model.add(layers.Dense(num_classes, activation="softmax"))
    # normalize output
    # model.add(layers.BatchNormalization())
    learning_rate = hp.Float("lr", min_value=1e-8, max_value=1e-1, sampling="log")
    model.compile(
        optimizer=keras.optimizers.SGD(learning_rate=polynomial),
        loss="categorical_crossentropy",
        metrics=["accuracy"],
    )
    return model

build_model(keras_tuner.HyperParameters())
```

Out[ ]:

```
<keras.engine.sequential.Sequential at 0x7f8132e7a970>
```

It turns out to have the best model score to be 0.728, with 3 hidden layers, as the first layer has 2976 neruals with activation function is `relu` , the second layer has 528 neurals, and the thrid layer has 1504 neruals. The learning rate is finally to be 0.01978.

In [ ]:

```
tuner = keras_tuner.RandomSearch(
    hypermodel=build_model,
    objective="val_accuracy",
    max_trials=3,
    executions_per_trial=3,
    overwrite=True,
    directory="/content/drive/MyDrive/HUDK_4050_Final/",
    project_name="tuner",
)

search_result = tuner.search(x = x_train, y = y_train, epochs = 40,

             batch_size = 128,
             validation_data = (x_test, y_test))

tuner.results_summary()
```

```
Trial 3 Complete [00h 14m 51s]
val_accuracy: 0.3615819215774536

Best val_accuracy So Far: 0.7288135488828024
Total elapsed time: 00h 44m 13s
Results summary
Results in /content/drive/MyDrive/HUDK_4050_Final/tuner
Showing 10 best trials
<keras_tuner.engine.objective.Objective object at 0x7f8138965160>
Trial summary
Hyperparameters:
1: 2976
activation: relu
num_layers: 2
units_0: 528
```

```
units_1: 1504
dropout: False
lr: 0.01978101759610635
Score: 0.7288135488828024
Trial summary
Hyperparameters:
1: 2176
activation: relu
num_layers: 5
units_0: 1472
units_1: 688
dropout: False
lr: 1.6997397224609708e-05
units_2: 944
units_3: 4048
units_4: 16
Score: 0.3615819215774536
Trial summary
Hyperparameters:
1: 1456
activation: relu
num_layers: 4
units_0: 3984
units_1: 1424
dropout: False
lr: 7.435257266599528e-05
units_2: 16
units_3: 16
Score: 0.35835350553194684
```

The final model has a 0.5544 as its accuracy score, and a loss of 866. Such huge lossess happen probabily because I set too many layer to try for tuner.

In [ ]:

```
# Get the top 3 hyperparameters.
best_hps = tuner.get_best_hyperparameters(3)
# Build the model with the best hp.
model = build_model(best_hps[0])
# Fit with the entire dataset.
# x_all = np.concatenate((x_train, x_test))
# y_all = np.concatenate((y_train, y_test))
# history = model.fit(x=x_all, y=y_all, epochs=10)
history = model.fit(x = x_train, y = y_train, epochs = 10)

score = model.evaluate(x_test, y_test, batch_size=32)
print("Network test score [loss, accuracy]:", score)

print(x_train.shape)
print(x_test.shape)
print(y_train.shape)
print(y_test.shape)
```

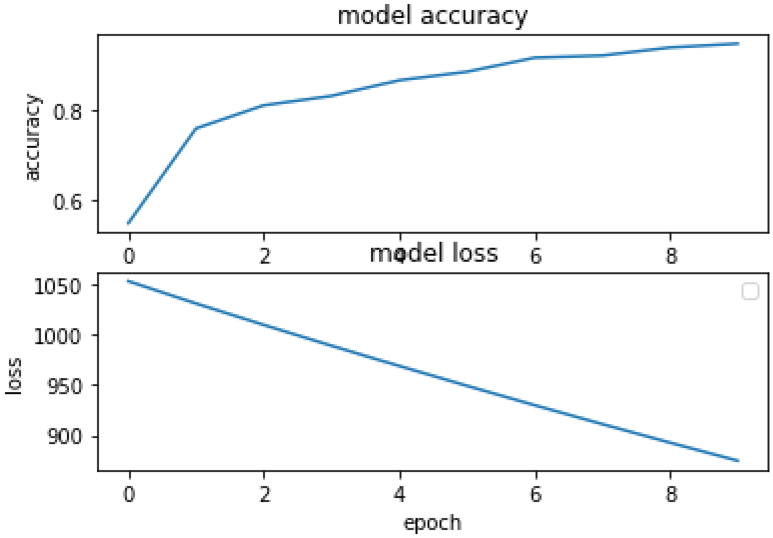
```
Epoch 1/10
52/52 [==============================] - 14s 256ms/step - loss: 1053.6150 - accuracy: 0.
5494
Epoch 2/10
52/52 [==============================] - 13s 257ms/step - loss: 1031.3456 - accuracy: 0.
7580
Epoch 3/10
52/52 [==============================] - 13s 253ms/step - loss: 1010.0507 - accuracy: 0.
8090
Epoch 4/10
52/52 [==============================] - 13s 250ms/step - loss: 989.3188 - accuracy: 0.8
296
```

```
Epoch 5/10
52/52 [==============================] - 13s 254ms/step - loss: 969.0494 - accuracy: 0.8
642
Epoch 6/10
52/52 [==============================] - 15s 282ms/step - loss: 949.2832 - accuracy: 0.8
830
Epoch 7/10
52/52 [==============================] - 13s 251ms/step - loss: 929.9527 - accuracy: 0.9
139
Epoch 8/10
52/52 [==============================] - 13s 253ms/step - loss: 911.1019 - accuracy: 0.9
187
Epoch 9/10
52/52 [==============================] - 13s 252ms/step - loss: 892.6476 - accuracy: 0.9
363
Epoch 10/10
52/52 [==============================] - 13s 249ms/step - loss: 874.6395 - accuracy: 0.9
448
13/13 [==============================] - 1s 96ms/step - loss: 866.6199 - accuracy: 0.554
5
Network test score [loss, accuracy]: [866.6199340820312, 0.5544794201850891]
(1649, 10000)
(413, 10000)
(1649, 10)
(413, 10)
```

In [ ]:

```
plt.subplot(2,1,1)
plt.plot(history.history['accuracy'])
plt.title('model accuracy')
plt.ylabel('accuracy')
plt.xlabel('epoch')


plt.subplot(2, 1, 2)
plt.plot(history.history["loss"])
plt.title("model loss")
plt.ylabel("loss")
plt.xlabel("epoch")
plt.legend()
plt.show()

plt.tight_layout()
```

```
WARNING:matplotlib.legend:No handles with labels found to put in legend.
```




```
<Figure size 432x288 with 0 Axes>
```

Discussion

The model is built from `keras.tuner` , which is to let the mechine ran all the data choices for me.

At first when I set up with my model, I had all the accuracy score as only 0.07 or 0.10. After then I tried to set more layers than people usually did and tried with more neruals as well. I found out that the accurary score is going up a little bit, but not too much.

I then started to use some regularization of L1/L2 and some dropouts and I found L2 makes my performance better. However, though it seems to be better, the accuracy score is still 0.20+.

Finally, I started to add up the epoch number, which makes the running time to be super big, and by training my dataset for several trials and epochs, I got my final accuracy to be 0.55. I think this might be a good accuracy score because the `Dense` layer is used to train the models that are not pictures. The `Dense` layeys are usually made to train these supervised data. If I have to raise my accuracy score to about 80 or 90, I probably should add some `max pooling1` layer into my tuner.

4.3 Convolutional Neural Networks

A convolutional neural network (CNN) is an artificial neural network commonly used in vision tasks. The use of convolutional layers allows CNNs to be very efficient regarding the amount of computation required, making them suitable for use on large datasets or devices with limited computing resources.

Data preparation

1. Import the required packages and set the class number as 10 and test size as 0.2.

In [ ]:
```
import os
import numpy as np
from os import listdir
from imageio import imread
from keras.utils import to_categorical
from sklearn.model_selection import train_test_split
from keras.utils.image_utils import img_to_array
import PIL
import matplotlib.pyplot as plt
from google.colab import drive
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras import layers
from tensorflow.keras.models import Sequential
from matplotlib import pyplot as plt

drive.mount('/content/drive')
# Settings
num_classes = 10
test_size = 0.2
```

```
Drive already mounted at /content/drive; to attempt to forcibly remount, call drive.moun
t("/content/drive", force_remount=True).
```

1. Get dataset from picture and then split to train and test set, and convert image to 3d array.

In [ ]:
```
def get_img(data_path):
  ## Getting image array from path:
  img = PIL.Image.open(data_path)
  img = img.convert("L")
  img = img_to_array(img)
  img = np.resize(img, (100, 100, 1))
  return img
```

In [ ]:
```
dataset_path = "/content/drive/MyDrive/4050final/Dataset"

## Getting all data from data path
labels = sorted(listdir(dataset_path))
X = []
Y = []
for i, label in enumerate(labels):
  data_path = dataset_path + "/" + label

  for data in listdir(data_path):
```

```
    img = get_img(data_path + "/" + data)
    X.append(img)
    Y.append(i)
## create dataset
X = 1 - np.array(X).astype("float32") /255
Y = np.array(Y).astype("float32")
Y = to_categorical(Y, num_classes)

X, X_test, Y, Y_test = train_test_split(X, Y, test_size=test_size, random_state = 42)
print(X.shape)
print(X_test.shape)
print(Y.shape)
print(Y_test.shape)
```

```
(130, 100, 100, 1)
(33, 100, 100, 1)
(130, 10)
(33, 10)
```

In [ ]:
```
#Unzip data
!unzip -q './drive/MyDrive/4050final/Dataset.zip'
```

```
replace __MACOSX/._Dataset? [y]es, [n]o, [A]ll, [N]one, [r]ename:
```

In [ ]:
```
img_height = 100
img_width = 100
batch_size = 128
```

1. Load data using Keras Utils so that they could be further used in the CNN model.

In [ ]:
```
#Load data using keras utils
train_ds = tf.keras.utils.image_dataset_from_directory(
    "Dataset",
    validation_split=0.2,
    subset="training",
    seed=1337,
    image_size=(img_height, img_width),
    batch_size=batch_size,
)

test_ds = tf.keras.utils.image_dataset_from_directory(
    "Dataset",
    validation_split=0.2,
    subset="validation",
    seed=1337,
    image_size=(img_height, img_width),
    batch_size=batch_size,
)
```

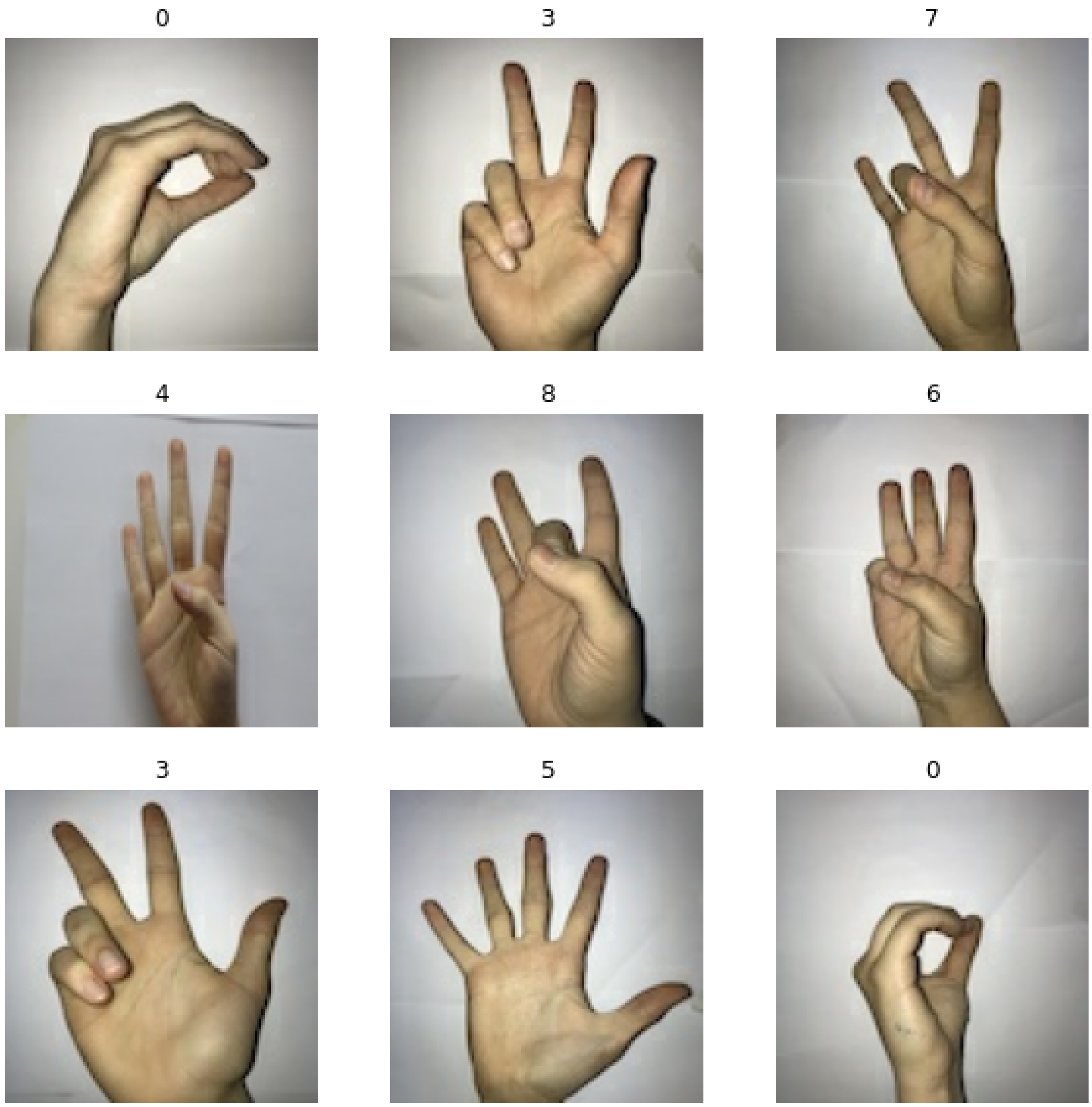
```
Found 2062 files belonging to 10 classes.
Using 1650 files for training.
Found 2062 files belonging to 10 classes.
Using 412 files for validation.
```

1. Check the class and image that to make sure they are prepared to be fitted in the model.

In [ ]:
```
#Print class names
class_names = train_ds.class_names
print(class_names)
```

```
['0', '1', '2', '3', '4', '5', '6', '7', '8', '9']
```

In [ ]:
```
#Plot images

plt.figure(figsize=(10, 10))
for images, labels in train_ds.take(1):
  for i in range(9):
    ax = plt.subplot(3, 3, i + 1)
    plt.imshow(images[i].numpy().astype("uint8"))
    plt.title(class_names[labels[i]])
    plt.axis("off")
```



Build model

1. Build an initial model. The first layer is a Rescaling layer that scales the input data by dividing each value by 255, a common preprocessing step that helps standardize the data. The next four layers are Conv2D, convolutional layers used for image classification. The activation argument specifies the activation function to use, in this case 'relu' (rectified linear unit). The model also includes four MaxPooling2D layers, which downsize the data by taking the maximum value over a certain window size specified by the pool_size argument. This helps reduce the size of the

data and can also help reduce overfitting. The Flatten layer flattens the data into a one-dimensional array, which is necessary before passing it through a Dense layer. The model has two Dense layers, fully connected (dense) layers that apply weights to the input data and produce an output. The first Dense layer has 128 units, and the second has num_classes units, the number of classes in the dataset. The final layer does not have an activation function, as it outputs the logits for each class.

In [ ]:
```
num_classes = len(class_names)
#Build model
model = Sequential([
  layers.Rescaling(1./255, input_shape=(img_height, img_width, 3)), #Standardize the da
  layers.Conv2D(16, 3, padding='same', activation='relu'),
  layers.MaxPooling2D(),
  layers.Conv2D(32, 3, padding='same', activation='relu'),
  layers.MaxPooling2D(),
  layers.Conv2D(64, 3, padding='same', activation='relu'),
  layers.MaxPooling2D(),
  layers.Flatten(),
  layers.Dense(128, activation='relu'),
  layers.Dense(num_classes)
])
```

1. In this case, the 'adam' optimizer is used, a popular choice for many tasks. And also, the SparseCategoricalCrossentropy loss function is used, a cross-entropy loss function suitable for multi-class classification tasks where the classes are mutually exclusive.

In [ ]:
```
#from_logits should be set to true since softmax was not used in the last output layer
#If softmax is used in the output layer then from_logits should be set to false (which
model.compile(optimizer='adam',
              loss=tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True),
              metrics=['accuracy'])
```

In [ ]:
```
model.summary()
```

```
Model: "sequential_2"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 rescaling_2 (Rescaling)     (None, 100, 100, 3)       0

 conv2d_6 (Conv2D)           (None, 100, 100, 16)      448

 max_pooling2d_6 (MaxPooling  (None, 50, 50, 16)       0
 2D)

 conv2d_7 (Conv2D)           (None, 50, 50, 32)        4640

 max_pooling2d_7 (MaxPooling  (None, 25, 25, 32)       0
 2D)

 conv2d_8 (Conv2D)           (None, 25, 25, 64)        18496

 max_pooling2d_8 (MaxPooling  (None, 12, 12, 64)       0
 2D)

 flatten_2 (Flatten)         (None, 9216)              0
```

```
 dense_4 (Dense)             (None, 128)               1179776

 dense_5 (Dense)             (None, 10)                1290

=================================================================
Total params: 1,204,650
Trainable params: 1,204,650
Non-trainable params: 0
_________________________________________________________________
```

Check the performance of the initial model and tuning parameters.

1. As the training progresses, we can see that the loss decreases and the accuracy increases for both the training and the validation data. This is a good sign that the model is learning and generalizing well to new data; after 20 epochs, the validation accuracy is 0.9053.

In [ ]:

```
epochs=20
history = model.fit(
  train_ds,
  validation_data=test_ds,
  epochs=epochs
)
```

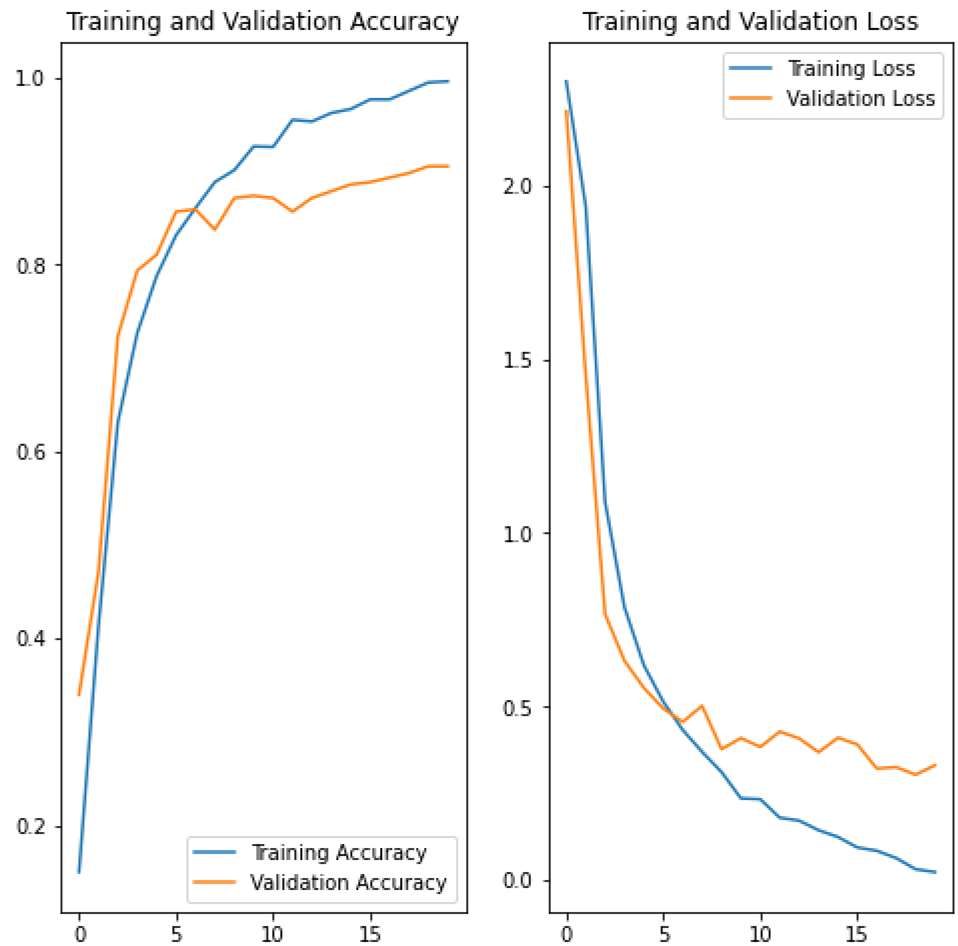
```
Epoch 1/20
13/13 [==============================] - 18s 1s/step - loss: 2.2991 - accuracy: 0.1497 -
val_loss: 2.2129 - val_accuracy: 0.3398
Epoch 2/20
13/13 [==============================] - 18s 1s/step - loss: 1.9405 - accuracy: 0.4127 -
val_loss: 1.4555 - val_accuracy: 0.4709
Epoch 3/20
13/13 [==============================] - 18s 1s/step - loss: 1.0894 - accuracy: 0.6309 -
val_loss: 0.7665 - val_accuracy: 0.7233
Epoch 4/20
13/13 [==============================] - 20s 2s/step - loss: 0.7858 - accuracy: 0.7267 -
val_loss: 0.6319 - val_accuracy: 0.7937
Epoch 5/20
13/13 [==============================] - 18s 1s/step - loss: 0.6190 - accuracy: 0.7879 -
val_loss: 0.5539 - val_accuracy: 0.8107
Epoch 6/20
13/13 [==============================] - 18s 1s/step - loss: 0.5141 - accuracy: 0.8315 -
val_loss: 0.4943 - val_accuracy: 0.8568
Epoch 7/20
13/13 [==============================] - 18s 1s/step - loss: 0.4328 - accuracy: 0.8606 -
val_loss: 0.4560 - val_accuracy: 0.8592
Epoch 8/20
13/13 [==============================] - 18s 1s/step - loss: 0.3697 - accuracy: 0.8885 -
val_loss: 0.5024 - val_accuracy: 0.8374
Epoch 9/20
13/13 [==============================] - 18s 1s/step - loss: 0.3116 - accuracy: 0.9012 -
val_loss: 0.3775 - val_accuracy: 0.8714
Epoch 10/20
13/13 [==============================] - 18s 1s/step - loss: 0.2359 - accuracy: 0.9267 -
val_loss: 0.4094 - val_accuracy: 0.8738
Epoch 11/20
13/13 [==============================] - 18s 1s/step - loss: 0.2333 - accuracy: 0.9261 -
val_loss: 0.3838 - val_accuracy: 0.8714
Epoch 12/20
13/13 [==============================] - 20s 1s/step - loss: 0.1801 - accuracy: 0.9552 -
val_loss: 0.4281 - val_accuracy: 0.8568
Epoch 13/20
13/13 [==============================] - 18s 1s/step - loss: 0.1717 - accuracy: 0.9533 -
val_loss: 0.4085 - val_accuracy: 0.8714
```

```
Epoch 14/20
13/13 [==============================] - 18s 1s/step - loss: 0.1442 - accuracy: 0.9624 -
val_loss: 0.3687 - val_accuracy: 0.8786
Epoch 15/20
13/13 [==============================] - 18s 1s/step - loss: 0.1246 - accuracy: 0.9667 -
val_loss: 0.4104 - val_accuracy: 0.8859
Epoch 16/20
13/13 [==============================] - 18s 1s/step - loss: 0.0944 - accuracy: 0.9770 -
val_loss: 0.3912 - val_accuracy: 0.8883
Epoch 17/20
13/13 [==============================] - 18s 1s/step - loss: 0.0850 - accuracy: 0.9770 -
val_loss: 0.3219 - val_accuracy: 0.8932
Epoch 18/20
13/13 [==============================] - 18s 1s/step - loss: 0.0639 - accuracy: 0.9861 -
val_loss: 0.3256 - val_accuracy: 0.8981
Epoch 19/20
13/13 [==============================] - 18s 1s/step - loss: 0.0319 - accuracy: 0.9952 -
val_loss: 0.3040 - val_accuracy: 0.9053
Epoch 20/20
13/13 [==============================] - 18s 1s/step - loss: 0.0235 - accuracy: 0.9964 -
val_loss: 0.3312 - val_accuracy: 0.9053
```

1. Then, we plot the training and validation accuracy and loss. We find that when the epoch is 7~8, the increase in validation accuracy starts to slow down significantly, and the decrease in validation loss also slows down significantly.

In [ ]:

```
#Plot training and test accuracy
acc = history.history['accuracy']
val_acc = history.history['val_accuracy']

loss = history.history['loss']
val_loss = history.history['val_loss']

epochs_range = range(epochs)

plt.figure(figsize=(8, 8))
plt.subplot(1, 2, 1)
plt.plot(epochs_range, acc, label='Training Accuracy')
plt.plot(epochs_range, val_acc, label='Validation Accuracy')
plt.legend(loc='lower right')
plt.title('Training and Validation Accuracy')

plt.subplot(1, 2, 2)
plt.plot(epochs_range, loss, label='Training Loss')
plt.plot(epochs_range, val_loss, label='Validation Loss')
plt.legend(loc='upper right')
plt.title('Training and Validation Loss')
plt.show()
```



In [ ]:
```
score = model.evaluate(test_ds)
print("Network test score [loss, accuracy]:", score)
```

```
4/4 [==============================] - 1s 279ms/step - loss: 0.4338 - accuracy: 0.8568
Network test score [loss, accuracy]: [0.4338451623916626, 0.856796145439148]
```

1. Then we do Adam with learning rate decay. we set the initial_learning_rate and decay_rate and decay step by ourselves. Using a learning rate schedule can help the model converge faster and can also help prevent overfitting. It allows the model to start with a larger learning rate and gradually reduce it as the training progresses. This can help the model escape from local minima and find a better solution.

In [ ]:
```
num_classes = len(class_names)
#Build model
model = Sequential([
  layers.Rescaling(1./255, input_shape=(img_height, img_width, 3)), #Standardize the da
  layers.Conv2D(16, 3, padding='same', activation='relu'),
  layers.MaxPooling2D(),
  layers.Conv2D(32, 3, padding='same', activation='relu'),
  layers.MaxPooling2D(),
  layers.Conv2D(64, 3, padding='same', activation='relu'),
  layers.MaxPooling2D(),
  layers.Flatten(),
  layers.Dense(128, activation='relu'),
  layers.Dense(num_classes)
])
```

In [ ]:
```
opt = keras.optimizers.Adam(learning_rate = keras.optimizers.schedules.ExponentialDecay
  initial_learning_rate = 5e-3,
  decay_rate = 0.96,
  decay_steps = 1500,
))
```

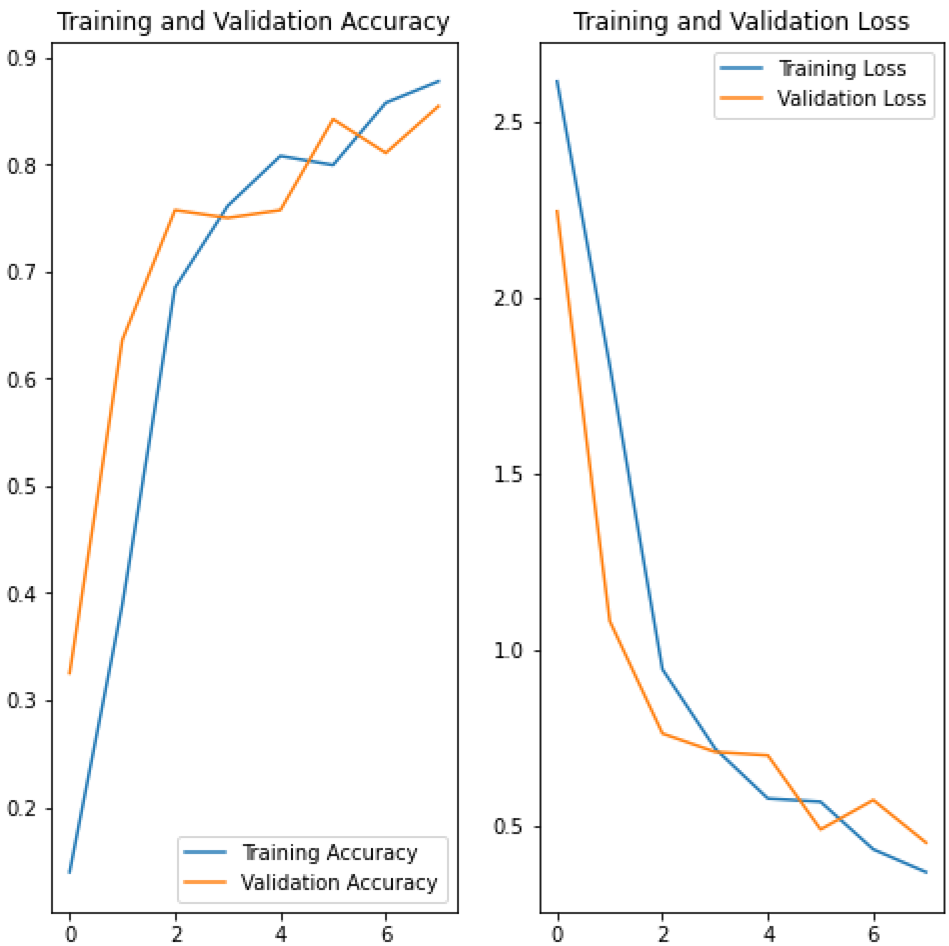
In [ ]:
```
model.compile(optimizer=opt,
              loss=tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True),
              metrics=['accuracy'])
```

In [ ]:
```
epochs=8
history = model.fit(
  train_ds,
  validation_data=test_ds,
  epochs=epochs
)
```

```
Epoch 1/8
13/13 [==============================] - 18s 1s/step - loss: 2.6153 - accuracy: 0.1394 -
val_loss: 2.2455 - val_accuracy: 0.3252
Epoch 2/8
13/13 [==============================] - 18s 1s/step - loss: 1.8071 - accuracy: 0.3891 -
val_loss: 1.0810 - val_accuracy: 0.6359
Epoch 3/8
13/13 [==============================] - 19s 1s/step - loss: 0.9437 - accuracy: 0.6848 -
val_loss: 0.7610 - val_accuracy: 0.7573
Epoch 4/8
13/13 [==============================] - 22s 2s/step - loss: 0.7181 - accuracy: 0.7612 -
val_loss: 0.7087 - val_accuracy: 0.7500
Epoch 5/8
13/13 [==============================] - 20s 1s/step - loss: 0.5762 - accuracy: 0.8079 -
val_loss: 0.6989 - val_accuracy: 0.7573
Epoch 6/8
13/13 [==============================] - 20s 2s/step - loss: 0.5669 - accuracy: 0.7994 -
val_loss: 0.4881 - val_accuracy: 0.8422
Epoch 7/8
13/13 [==============================] - 18s 1s/step - loss: 0.4315 - accuracy: 0.8576 -
val_loss: 0.5718 - val_accuracy: 0.8107
Epoch 8/8
13/13 [==============================] - 17s 1s/step - loss: 0.3667 - accuracy: 0.8776 -
val_loss: 0.4507 - val_accuracy: 0.8544
```

In [ ]:
```
#Plot training and test accuracy
acc = history.history['accuracy']
val_acc = history.history['val_accuracy']

loss = history.history['loss']
val_loss = history.history['val_loss']

epochs_range = range(epochs)

plt.figure(figsize=(8, 8))
plt.subplot(1, 2, 1)
plt.plot(epochs_range, acc, label='Training Accuracy')
plt.plot(epochs_range, val_acc, label='Validation Accuracy')
plt.legend(loc='lower right')
plt.title('Training and Validation Accuracy')

plt.subplot(1, 2, 2)
plt.plot(epochs_range, loss, label='Training Loss')
plt.plot(epochs_range, val_loss, label='Validation Loss')
```

```
plt.legend(loc='upper right')
plt.title('Training and Validation Loss')
plt.show()
```



1. Checking the loss curve, we found that the fluctuation of the validation curve decreased, so we further reduced the learning rate

In [ ]:
```
num_classes = len(class_names)
#Build model
model = Sequential([
  layers.Rescaling(1./255, input_shape=(img_height, img_width, 3)), #Standardize the da
  layers.Conv2D(16, 3, padding='same', activation='relu'),
  layers.MaxPooling2D(),
  layers.Conv2D(32, 3, padding='same', activation='relu'),
  layers.MaxPooling2D(),
  layers.Conv2D(64, 3, padding='same', activation='relu'),
  layers.MaxPooling2D(),
  layers.Flatten(),
  layers.Dense(128, activation='relu'),
  layers.Dense(num_classes)
])
```

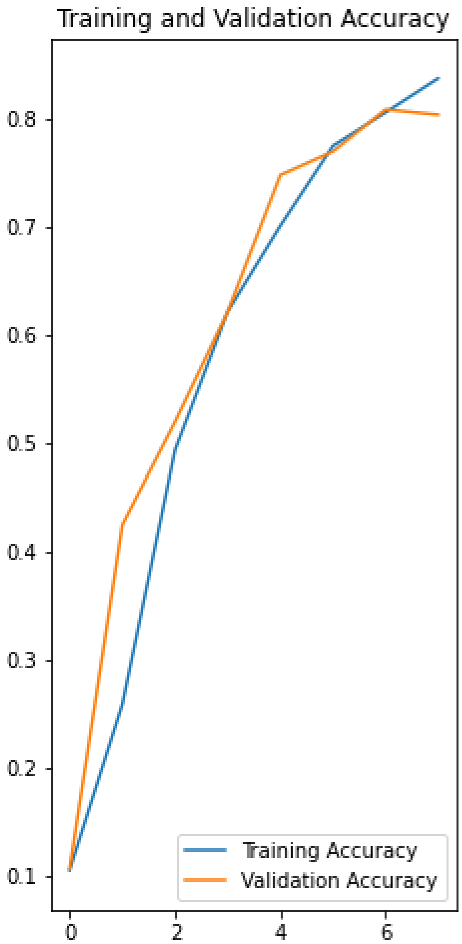
In [ ]:
```
opt = keras.optimizers.Adam(learning_rate = keras.optimizers.schedules.ExponentialDecay
  initial_learning_rate = 5e-4,
  decay_rate = 0.96,
  decay_steps = 1500,
))
```

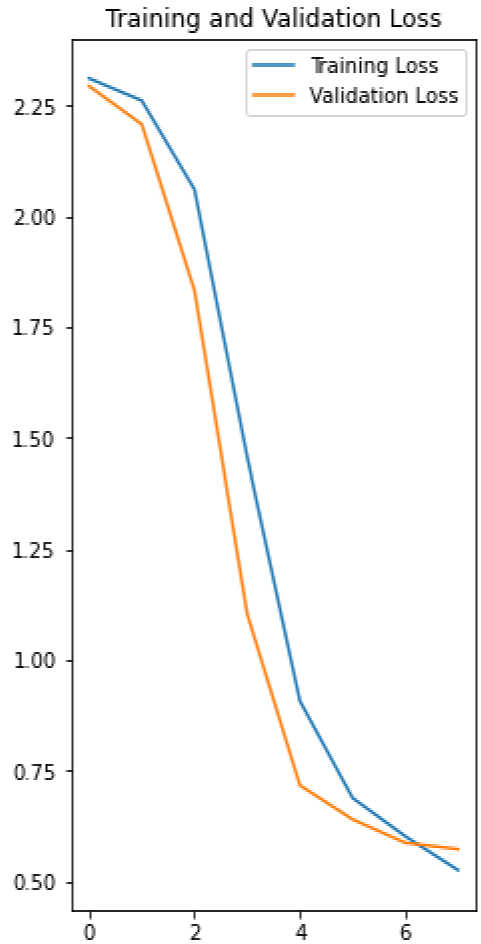
In [ ]:
```
model.compile(optimizer=opt,
              loss=tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True),
              metrics=['accuracy'])
```

In [ ]:
```
epochs = 8
history = model.fit(
  train_ds,
  validation_data=test_ds,
  epochs=epochs
)
```

```
Epoch 1/8
13/13 [==============================] - 19s 1s/step - loss: 2.3116 - accuracy: 0.1055 -
val_loss: 2.2941 - val_accuracy: 0.1068
Epoch 2/8
13/13 [==============================] - 20s 1s/step - loss: 2.2615 - accuracy: 0.2588 -
val_loss: 2.2078 - val_accuracy: 0.4248
Epoch 3/8
13/13 [==============================] - 21s 2s/step - loss: 2.0607 - accuracy: 0.4933 -
val_loss: 1.8326 - val_accuracy: 0.5194
Epoch 4/8
13/13 [==============================] - 18s 1s/step - loss: 1.4571 - accuracy: 0.6212 -
val_loss: 1.1043 - val_accuracy: 0.6214
Epoch 5/8
13/13 [==============================] - 20s 1s/step - loss: 0.9077 - accuracy: 0.7006 -
val_loss: 0.7173 - val_accuracy: 0.7476
Epoch 6/8
13/13 [==============================] - 19s 1s/step - loss: 0.6886 - accuracy: 0.7745 -
val_loss: 0.6401 - val_accuracy: 0.7694
Epoch 7/8
13/13 [==============================] - 18s 1s/step - loss: 0.6020 - accuracy: 0.8055 -
val_loss: 0.5873 - val_accuracy: 0.8083
Epoch 8/8
13/13 [==============================] - 18s 1s/step - loss: 0.5257 - accuracy: 0.8370 -
val_loss: 0.5730 - val_accuracy: 0.8034
```

In [ ]:
```
#Plot training and test accuracy
acc = history.history['accuracy']
val_acc = history.history['val_accuracy']

loss = history.history['loss']
val_loss = history.history['val_loss']

epochs_range = range(epochs)

plt.figure(figsize=(8, 8))
plt.subplot(1, 2, 1)
plt.plot(epochs_range, acc, label='Training Accuracy')
plt.plot(epochs_range, val_acc, label='Validation Accuracy')
plt.legend(loc='lower right')
plt.title('Training and Validation Accuracy')

plt.subplot(1, 2, 2)
plt.plot(epochs_range, loss, label='Training Loss')
plt.plot(epochs_range, val_loss, label='Validation Loss')
plt.legend(loc='upper right')
plt.title('Training and Validation Loss')
plt.show()
```





This time, we saw a smooth decline in the loss curve, indicating that our learning rate was selected appropriately, and the accuracy of the training set was not much different from that of the test set. Therefore, the parameter tuning of the model comes to an end. The accuracy of the test set is 0.803.

In [ ]:

```
score = model.evaluate(test_ds)
print("Network test score [loss, accuracy]:", score)
```

```
4/4 [==============================] - 1s 271ms/step - loss: 0.5730 - accuracy: 0.8034
Network test score [loss, accuracy]: [0.5729589462280273, 0.803398072719574]
```

4.4 VGG16

Data Preprocess of VGG16

In [ ]:
```
import os
import numpy as np
from os import listdir
from imageio import imread
from keras.utils import to_categorical
from sklearn.model_selection import train_test_split
from keras.utils.image_utils import img_to_array
import keras
import PIL
import matplotlib.pyplot as plt
```

In [ ]:
```
# Settings
num_classes = 10
test_size = 0.2
```

Read Image and Convert to 3D Array

In [ ]:
```
def get_img(data_path):
  ## Getting image array from path:
  img = PIL.Image.open(data_path)
  img = img.convert("L")
  img = img_to_array(img)
  img = np.resize(img, (100, 100, 3))
  return img
```

Get dataset from picture and then split to train and test set

In [ ]:
```
from google.colab import drive
drive.mount('/content/drive')
dataset_path = "/content/drive/MyDrive/Dataset"

## Getting all data from data path
labels = sorted(listdir(dataset_path))
X = []
Y = []
for i, label in enumerate(labels):
  data_path = dataset_path + "/" + label

  for data in listdir(data_path):
    img = get_img(data_path + "/" + data)
    X.append(img)
    Y.append(i)
## create dataset
X = 1 - np.array(X).astype("float32") /255
Y = np.array(Y).astype("float32")
Y = to_categorical(Y, num_classes)

X, X_test, Y, Y_test = train_test_split(X, Y, test_size=test_size, random_state = 42)
print(X.shape)
print(X_test.shape)
print(Y.shape)
print(Y_test.shape)
```

```
Drive already mounted at /content/drive; to attempt to forcibly remount, call drive.moun
t("/content/drive", force_remount=True).
(1649, 100, 100, 3)
(413, 100, 100, 3)
(1649, 10)
(413, 10)
```

In [ ]:
```
import tensorflow as tf
from numpy.random import seed
seed(1)
tf.random.set_seed(123)
```

In [ ]:
```
import tensorflow as tf
from tensorflow import keras
import numpy as np
import pandas as pd
import sklearn as sk
import time
from keras.datasets import mnist
from keras.models import Sequential, load_model
from keras.layers import Dense, Dropout, Flatten
from keras import optimizers
from keras import backend as K
from keras import regularizers
from keras import initializers
from matplotlib import pyplot as plt
from sklearn.model_selection import train_test_split
from keras.utils import to_categorical
import math
from keras import applications
```

In [ ]:
```
img_height = 100
img_width = 100
```

VGG16

Learning rate has been adjusted between le-3, le-4, le-5, le-6, the result is le-5 can provide the best performance. Drop out rate of 0.3, 0.4, 0.5 has been tried and ultimately 0.4 has a relativley good performance. Initially epoch has been set to 10 but the performance was not pretty well. By increasing to 50, the model has been trained into the accuracy of 98% GlobalAveragePooling2D()for creating feature map for each category of the model and unfreezing the base model and retrain the whole model for fine-tuning has been applied in all transfer learning model.

In [ ]:
```
from tensorflow.keras.applications import VGG16
#base_model = VGG16(include_top = False, weights = 'imagenet',input_shape=(100,100,3))
base_model = keras.applications.VGG16(
    weights='imagenet',
    input_shape=(img_height, img_width, 3),
    include_top=False)
#base_model.summary()
```

In [ ]:
```
base_model.trainable = False
```

In [ ]:
```
inputs = keras.Input(shape=(img_height, img_width, 3))
```

In [ ]:
```
x=tf.keras.applications.vgg16.preprocess_input(
    inputs, data_format=None
)
```

In [ ]:
```
x = base_model(x, training=False)
x = keras.layers.GlobalAveragePooling2D()(x)
x = keras.layers.Dropout(0.4)(x)
outputs = keras.layers.Dense(10)(x)

model = keras.Model(inputs, outputs)
```

In [ ]:
```
model.summary()
```

```
Model: "model_3"
_________________________________________________________________
 Layer (type)                Output Shape              Param #   
=================================================================
 input_12 (InputLayer)       [(None, 100, 100, 3)]     0         
                                                                 
 tf.__operators__.getitem_4   (None, 100, 100, 3)      0         
 (SlicingOpLambda)                                               
                                                                 
 tf.nn.bias_add_4 (TFOpLambd  (None, 100, 100, 3)      0         
 a)                                                              
                                                                 
 vgg16 (Functional)          (None, 3, 3, 512)         14714688  
                                                                 
 global_average_pooling2d_4   (None, 512)              0         
 (GlobalAveragePooling2D)                                        
                                                                 
 dropout_3 (Dropout)         (None, 512)               0         
                                                                 
 dense_3 (Dense)             (None, 10)                5130      
                                                                 
=================================================================
Total params: 14,719,818
Trainable params: 5,130
Non-trainable params: 14,714,688
_________________________________________________________________
```

In [ ]:
```
model.compile(optimizer='adam',
              loss=tf.keras.losses.CategoricalCrossentropy(from_logits=True),
              metrics=['accuracy'])
model.fit(X, Y, epochs=3, validation_data=(X_test,Y_test))
```

```
Epoch 1/3
52/52 [==============================] - 4s 58ms/step - loss: 4.3663 - accuracy: 0.1049
- val_loss: 2.3922 - val_accuracy: 0.0969
Epoch 2/3
52/52 [==============================] - 2s 48ms/step - loss: 2.8542 - accuracy: 0.1104
- val_loss: 2.3356 - val_accuracy: 0.0847
Epoch 3/3
52/52 [==============================] - 3s 48ms/step - loss: 2.5525 - accuracy: 0.1013
- val_loss: 2.3267 - val_accuracy: 0.0847
```

Out[ ]: `<keras.callbacks.History at 0x7fbe0732ae20>`

In [ ]:
```
# fine-tuning
base_model.trainable = True
model.summary()

model.compile(
```

```
    optimizer=keras.optimizers.Adam(1e-5),  # Low learning rate
    loss=keras.losses.CategoricalCrossentropy(from_logits=True),
    metrics=['accuracy']
)

epochs = 50
model.fit(X, Y, epochs=epochs, validation_data=(X_test,Y_test))
```

```
Model: "model_3"
_________________________________________________________________
 Layer (type)                Output Shape              Param #   
=================================================================
 input_12 (InputLayer)       [(None, 100, 100, 3)]     0         
                                                                 
 tf.__operators__.getitem_4   (None, 100, 100, 3)      0         
 (SlicingOpLambda)                                               
                                                                 
 tf.nn.bias_add_4 (TFOpLambd  (None, 100, 100, 3)      0         
 a)                                                              
                                                                 
 vgg16 (Functional)          (None, 3, 3, 512)         14714688  
                                                                 
 global_average_pooling2d_4   (None, 512)              0         
 (GlobalAveragePooling2D)                                        
                                                                 
 dropout_3 (Dropout)         (None, 512)               0         
                                                                 
 dense_3 (Dense)             (None, 10)                5130      
                                                                 
=================================================================
Total params: 14,719,818
Trainable params: 14,719,818
Non-trainable params: 0
_________________________________________________________________
Epoch 1/50
52/52 [==============================] - 7s 121ms/step - loss: 2.3724 - accuracy: 0.0958
- val_loss: 2.3056 - val_accuracy: 0.1211
Epoch 2/50
52/52 [==============================] - 6s 117ms/step - loss: 2.3099 - accuracy: 0.1079
- val_loss: 2.2978 - val_accuracy: 0.1162
Epoch 3/50
52/52 [==============================] - 6s 118ms/step - loss: 2.3116 - accuracy: 0.1031
- val_loss: 2.2990 - val_accuracy: 0.1356
Epoch 4/50
52/52 [==============================] - 6s 119ms/step - loss: 2.3058 - accuracy: 0.1164
- val_loss: 2.2979 - val_accuracy: 0.0920
Epoch 5/50
52/52 [==============================] - 6s 121ms/step - loss: 2.2945 - accuracy: 0.1164
- val_loss: 2.3042 - val_accuracy: 0.0775
Epoch 6/50
52/52 [==============================] - 6s 122ms/step - loss: 2.2873 - accuracy: 0.1128
- val_loss: 2.2932 - val_accuracy: 0.1525
Epoch 7/50
52/52 [==============================] - 6s 121ms/step - loss: 2.2633 - accuracy: 0.1407
- val_loss: 2.2473 - val_accuracy: 0.1550
Epoch 8/50
52/52 [==============================] - 6s 119ms/step - loss: 2.1726 - accuracy: 0.1686
- val_loss: 1.9856 - val_accuracy: 0.2373
Epoch 9/50
52/52 [==============================] - 6s 119ms/step - loss: 1.9864 - accuracy: 0.2608
- val_loss: 1.8227 - val_accuracy: 0.2421
Epoch 10/50
52/52 [==============================] - 6s 118ms/step - loss: 1.4740 - accuracy: 0.4463
- val_loss: 0.9207 - val_accuracy: 0.6998
Epoch 11/50
```

```
52/52 [==============================] - 6s 118ms/step - loss: 1.1139 - accuracy: 0.5919
- val_loss: 0.8084 - val_accuracy: 0.7191
Epoch 12/50
52/52 [==============================] - 6s 118ms/step - loss: 0.8675 - accuracy: 0.6871
- val_loss: 0.6650 - val_accuracy: 0.7554
Epoch 13/50
52/52 [==============================] - 6s 118ms/step - loss: 0.7169 - accuracy: 0.7374
- val_loss: 0.5854 - val_accuracy: 0.8160
Epoch 14/50
52/52 [==============================] - 6s 118ms/step - loss: 0.6123 - accuracy: 0.7829
- val_loss: 0.3710 - val_accuracy: 0.8765
Epoch 15/50
52/52 [==============================] - 6s 119ms/step - loss: 0.4866 - accuracy: 0.8369
- val_loss: 0.4846 - val_accuracy: 0.8450
Epoch 16/50
52/52 [==============================] - 6s 121ms/step - loss: 0.5350 - accuracy: 0.8114
- val_loss: 0.5098 - val_accuracy: 0.8232
Epoch 17/50
52/52 [==============================] - 6s 120ms/step - loss: 0.4257 - accuracy: 0.8466
- val_loss: 0.4723 - val_accuracy: 0.8499
Epoch 18/50
52/52 [==============================] - 6s 120ms/step - loss: 0.3911 - accuracy: 0.8629
- val_loss: 0.3960 - val_accuracy: 0.8741
Epoch 19/50
52/52 [==============================] - 6s 121ms/step - loss: 0.3198 - accuracy: 0.8896
- val_loss: 0.2608 - val_accuracy: 0.9249
Epoch 20/50
52/52 [==============================] - 6s 121ms/step - loss: 0.3174 - accuracy: 0.8896
- val_loss: 0.3662 - val_accuracy: 0.8789
Epoch 21/50
52/52 [==============================] - 6s 121ms/step - loss: 0.3209 - accuracy: 0.8890
- val_loss: 0.2470 - val_accuracy: 0.9322
Epoch 22/50
52/52 [==============================] - 6s 120ms/step - loss: 0.2785 - accuracy: 0.9078
- val_loss: 0.2448 - val_accuracy: 0.9298
Epoch 23/50
52/52 [==============================] - 6s 120ms/step - loss: 0.1746 - accuracy: 0.9351
- val_loss: 0.2101 - val_accuracy: 0.9443
Epoch 24/50
52/52 [==============================] - 6s 120ms/step - loss: 0.2193 - accuracy: 0.9200
- val_loss: 0.2816 - val_accuracy: 0.9153
Epoch 25/50
52/52 [==============================] - 6s 120ms/step - loss: 0.2029 - accuracy: 0.9303
- val_loss: 0.1815 - val_accuracy: 0.9540
Epoch 26/50
52/52 [==============================] - 6s 120ms/step - loss: 0.1532 - accuracy: 0.9472
- val_loss: 0.2500 - val_accuracy: 0.9274
Epoch 27/50
52/52 [==============================] - 6s 119ms/step - loss: 0.1755 - accuracy: 0.9466
- val_loss: 0.2564 - val_accuracy: 0.9346
Epoch 28/50
52/52 [==============================] - 6s 119ms/step - loss: 0.1698 - accuracy: 0.9388
- val_loss: 0.1778 - val_accuracy: 0.9467
Epoch 29/50
52/52 [==============================] - 6s 119ms/step - loss: 0.1109 - accuracy: 0.9636
- val_loss: 0.2167 - val_accuracy: 0.9492
Epoch 30/50
52/52 [==============================] - 6s 120ms/step - loss: 0.1336 - accuracy: 0.9521
- val_loss: 0.2513 - val_accuracy: 0.9225
Epoch 31/50
52/52 [==============================] - 6s 119ms/step - loss: 0.1439 - accuracy: 0.9515
- val_loss: 0.1878 - val_accuracy: 0.9564
Epoch 32/50
52/52 [==============================] - 6s 120ms/step - loss: 0.1092 - accuracy: 0.9630
- val_loss: 0.2315 - val_accuracy: 0.9370
```

```
Epoch 33/50
52/52 [==============================] - 6s 120ms/step - loss: 0.0907 - accuracy: 0.9685
- val_loss: 0.2425 - val_accuracy: 0.9370
Epoch 34/50
52/52 [==============================] - 6s 122ms/step - loss: 0.1213 - accuracy: 0.9576
- val_loss: 0.2830 - val_accuracy: 0.9201
Epoch 35/50
52/52 [==============================] - 6s 121ms/step - loss: 0.1059 - accuracy: 0.9630
- val_loss: 0.2733 - val_accuracy: 0.9298
Epoch 36/50
52/52 [==============================] - 6s 120ms/step - loss: 0.1107 - accuracy: 0.9594
- val_loss: 0.1939 - val_accuracy: 0.9492
Epoch 37/50
52/52 [==============================] - 6s 120ms/step - loss: 0.0740 - accuracy: 0.9697
- val_loss: 0.1391 - val_accuracy: 0.9661
Epoch 38/50
52/52 [==============================] - 6s 119ms/step - loss: 0.0795 - accuracy: 0.9721
- val_loss: 0.1908 - val_accuracy: 0.9467
Epoch 39/50
52/52 [==============================] - 6s 120ms/step - loss: 0.0686 - accuracy: 0.9745
- val_loss: 0.2074 - val_accuracy: 0.9492
Epoch 40/50
52/52 [==============================] - 6s 120ms/step - loss: 0.0757 - accuracy: 0.9715
- val_loss: 0.2020 - val_accuracy: 0.9467
Epoch 41/50
52/52 [==============================] - 6s 123ms/step - loss: 0.0877 - accuracy: 0.9691
- val_loss: 0.1446 - val_accuracy: 0.9516
Epoch 42/50
52/52 [==============================] - 6s 120ms/step - loss: 0.0482 - accuracy: 0.9830
- val_loss: 0.1810 - val_accuracy: 0.9564
Epoch 43/50
52/52 [==============================] - 6s 120ms/step - loss: 0.0632 - accuracy: 0.9770
- val_loss: 0.1555 - val_accuracy: 0.9564
Epoch 44/50
52/52 [==============================] - 6s 120ms/step - loss: 0.0484 - accuracy: 0.9848
- val_loss: 0.1387 - val_accuracy: 0.9661
Epoch 45/50
52/52 [==============================] - 6s 120ms/step - loss: 0.0437 - accuracy: 0.9854
- val_loss: 0.3471 - val_accuracy: 0.8959
Epoch 46/50
52/52 [==============================] - 6s 122ms/step - loss: 0.0600 - accuracy: 0.9794
- val_loss: 0.1436 - val_accuracy: 0.9661
Epoch 47/50
52/52 [==============================] - 6s 120ms/step - loss: 0.0369 - accuracy: 0.9861
- val_loss: 0.2532 - val_accuracy: 0.9298
Epoch 48/50
52/52 [==============================] - 6s 119ms/step - loss: 0.0545 - accuracy: 0.9806
- val_loss: 0.1625 - val_accuracy: 0.9637
Epoch 49/50
52/52 [==============================] - 6s 120ms/step - loss: 0.0461 - accuracy: 0.9861
- val_loss: 0.1833 - val_accuracy: 0.9613
Epoch 50/50
52/52 [==============================] - 6s 120ms/step - loss: 0.0377 - accuracy: 0.9873
- val_loss: 0.1540 - val_accuracy: 0.9564
```

Out[ ]:
```
<keras.callbacks.History at 0x7fbe075a49d0>
```

Accuracy: 98.73%

By comparing model of transfered learning model with Based on the results from model we built from scratch, the accuracy of fully connected model is 71%, accuracy of CNN model is 80.34%. All the five models of transfer learning have a better performance than those two since transfer learning will include the saving of resources and improve efficiety when training new models with complex layers.

4.5 X-ception, ResNet50, Inceptionv3

Data Preprocess of X-ception, ResNet50, Inceptionv3

In [ ]:
```
import os
import numpy as np
from os import listdir
from imageio import imread
from keras.utils import to_categorical
from sklearn.model_selection import train_test_split
from keras.utils.image_utils import img_to_array

import PIL
import matplotlib.pyplot as plt
```

In [ ]:
```
# Settings
num_classes = 10
test_size = 0.2
```

read image and convert to 3d array

In [ ]:
```
def get_img(data_path):
  ## Getting image array from path:
  img = PIL.Image.open(data_path)
  img = img.convert("L")
  img = img_to_array(img)
  img = np.resize(img, (100, 100, 3))
  return img
```

Get dataset from picture and then split to train and test set

In [ ]:
```
from google.colab import drive
drive.mount('/content/drive')
dataset_path = "/content/drive/MyDrive/Dataset"

## Getting all data from data path
labels = sorted(listdir(dataset_path))
X = []
Y = []
for i, label in enumerate(labels):
  data_path = dataset_path + "/" + label

  for data in listdir(data_path):
    img = get_img(data_path + "/" + data)
    X.append(img)
    Y.append(i)
## create dataset
X = 1 - np.array(X).astype("float32") /255
Y = np.array(Y).astype("float32")
Y = to_categorical(Y, num_classes)

X, X_test, Y, Y_test = train_test_split(X, Y, test_size=test_size, random_state = 42)
print(X.shape)
print(X_test.shape)
print(Y.shape)
print(Y_test.shape)
```

```
Mounted at /content/drive
(1649, 100, 100, 3)
(413, 100, 100, 3)
(1649, 10)
(413, 10)
```

In [ ]:
```
import tensorflow as tf
from numpy.random import seed
seed(123)
tf.random.set_seed(123)
```

In [ ]:
```
import tensorflow as tf
from tensorflow import keras
import numpy as np
import pandas as pd
import sklearn as sk
import time
from keras.datasets import mnist
from keras.models import Sequential, load_model
from keras.layers import Dense, Dropout, Flatten
from keras import optimizers
from keras import backend as K
from keras import regularizers
from keras import initializers
from matplotlib import pyplot as plt
from sklearn.model_selection import train_test_split
from keras.utils import to_categorical
import math
from keras import applications
```

In [ ]:
```
img_height = 100
img_width = 100
```

In [ ]:
```
# Creating validation set and training set by partitioning the current training set
val = X[:274]
partial = X[274:]
val_labels = Y[:274]
partial_labels = Y[274:]
```

In [ ]:
```
print(X.shape)
print(val.shape)
print(partial.shape)
```

```
(1649, 100, 100, 3)
(274, 100, 100, 3)
(1375, 100, 100, 3)
```

X-ception

When building the last layers of X-ception, I first added the GlobalAveragePooling2D() to create feature map for each cagetory. I then added dense layer but it didn't help. I tried several drop out values and found 0.4 the best. After tuning the last layers, I unfreeze the base model and retrain the whole model with a very low learning rate. I've tried some different values of learning rate and found lr = le-5 the best. When fit the model, I used EarlyStopping function in keras to find the optimal epoch value (=27) to avoid the issue of overfiffting.

In [ ]:
```
#Load the Xception pre-trained model
#include_top=False means that you’re not interested in the last layer of the model. You
base_model = keras.applications.Xception(
    weights='imagenet',
    input_shape=(img_height, img_width, 3),
    include_top=False)
```

In [ ]:
```
#To prevent the base model being retrained
base_model.trainable = False
```

In [ ]:
```
inputs = keras.Input(shape=(img_height, img_width, 3))
```

In [ ]:
```
#Preprocess inputs as expected by Xception
#scale from (0,1) to (-1,1)
x = tf.keras.applications.xception.preprocess_input(inputs)
```

In [ ]:
```
#Build the last layers
#Use the functional API method in Keras to illustrate this approach
x = base_model(x, training=False)
x = keras.layers.GlobalAveragePooling2D()(x)
x = keras.layers.Dropout(0.4)(x)
outputs = keras.layers.Dense(10)(x)
model = keras.Model(inputs, outputs)
```

In [ ]:
```
model.summary()
```

```
Model: "model_16"
_________________________________________________________________
 Layer (type)                Output Shape              Param #   
=================================================================
 input_36 (InputLayer)       [(None, 100, 100, 3)]     0         
                                                                 
 tf.math.truediv_17 (TFOpLam  (None, 100, 100, 3)      0         
 bda)                                                            
                                                                 
 tf.math.subtract_17 (TFOpLa  (None, 100, 100, 3)      0         
 mbda)                                                           
                                                                 
 xception (Functional)       (None, 3, 3, 2048)        20861480  
                                                                 
 global_average_pooling2d_16  (None, 2048)             0         
  (GlobalAveragePooling2D)                                       
                                                                 
 dropout_16 (Dropout)        (None, 2048)              0         
                                                                 
 dense_34 (Dense)            (None, 10)                20490     
                                                                 
=================================================================
Total params: 20,881,970
Trainable params: 20,490
Non-trainable params: 20,861,480
_________________________________________________________________
```

In [ ]:
```
model.compile(optimizer='adam',
              loss=tf.keras.losses.CategoricalCrossentropy(from_logits=True),
              metrics=['accuracy'])
model.fit(X, Y, epochs=3, validation_data=(X_test,Y_test))
```

```
Epoch 1/3
52/52 [==============================] - 5s 58ms/step - loss: 2.2971 - accuracy: 0.1055
```

```
- val_loss: 2.2941 - val_accuracy: 0.1259
Epoch 2/3
52/52 [==============================] - 2s 37ms/step - loss: 2.2825 - accuracy: 0.1686
- val_loss: 2.2829 - val_accuracy: 0.1453
Epoch 3/3
52/52 [==============================] - 2s 37ms/step - loss: 2.2684 - accuracy: 0.1740
- val_loss: 2.2713 - val_accuracy: 0.1743
```

Out[ ]:
```
<keras.callbacks.History at 0x7fd436c87d90>
```

In [ ]:
```python
# Fine-tuning
base_model.trainable = True
model.summary()

model.compile(
    optimizer=keras.optimizers.Adam(1e-5),  # Low learning rate
    loss=keras.losses.CategoricalCrossentropy(from_logits=True),
    metrics=['accuracy']
)
```

```
Model: "model_16"
_________________________________________________________________
 Layer (type)                Output Shape              Param #   
=================================================================
 input_36 (InputLayer)       [(None, 100, 100, 3)]     0         
                                                                 
 tf.math.truediv_17 (TFOpLam  (None, 100, 100, 3)      0         
 bda)                                                            
                                                                 
 tf.math.subtract_17 (TFOpLa  (None, 100, 100, 3)      0         
 mbda)                                                           
                                                                 
 xception (Functional)       (None, 3, 3, 2048)        20861480  
                                                                 
 global_average_pooling2d_16  (None, 2048)             0         
  (GlobalAveragePooling2D)                                       
                                                                 
 dropout_16 (Dropout)        (None, 2048)              0         
                                                                 
 dense_34 (Dense)            (None, 10)                20490     
                                                                 
=================================================================
Total params: 20,881,970
Trainable params: 20,827,442
Non-trainable params: 54,528
_________________________________________________________________
```

In [ ]:
```python
from keras import callbacks
earlystopping = callbacks.EarlyStopping(monitor ="val_loss",
                                        mode ="min", patience = 5,
                                        restore_best_weights = True)

history = model.fit(partial, partial_labels, batch_size = 16,
                    epochs = 100, validation_data =(val, val_labels),
                    callbacks =[earlystopping])
```

```
Epoch 1/100
86/86 [==============================] - 11s 84ms/step - loss: 2.1110 - accuracy: 0.1927
- val_loss: 1.8732 - val_accuracy: 0.2920
Epoch 2/100
86/86 [==============================] - 7s 85ms/step - loss: 1.7513 - accuracy: 0.3236
- val_loss: 1.7132 - val_accuracy: 0.2993
Epoch 3/100
86/86 [==============================] - 7s 78ms/step - loss: 1.4120 - accuracy: 0.4465
```

```
- val_loss: 1.4318 - val_accuracy: 0.4234
Epoch 4/100
86/86 [==============================] - 7s 77ms/step - loss: 1.2753 - accuracy: 0.4996
- val_loss: 1.2722 - val_accuracy: 0.5182
Epoch 5/100
86/86 [==============================] - 6s 73ms/step - loss: 1.1707 - accuracy: 0.5462
- val_loss: 1.1859 - val_accuracy: 0.5803
Epoch 6/100
86/86 [==============================] - 6s 74ms/step - loss: 1.1123 - accuracy: 0.5862
- val_loss: 1.0707 - val_accuracy: 0.5876
Epoch 7/100
86/86 [==============================] - 6s 73ms/step - loss: 1.0861 - accuracy: 0.5876
- val_loss: 1.2931 - val_accuracy: 0.4708
Epoch 8/100
86/86 [==============================] - 6s 74ms/step - loss: 1.0506 - accuracy: 0.6065
- val_loss: 0.9768 - val_accuracy: 0.6496
Epoch 9/100
86/86 [==============================] - 6s 74ms/step - loss: 0.9626 - accuracy: 0.6429
- val_loss: 1.3748 - val_accuracy: 0.4964
Epoch 10/100
86/86 [==============================] - 6s 74ms/step - loss: 0.8996 - accuracy: 0.6633
- val_loss: 0.9462 - val_accuracy: 0.6533
Epoch 11/100
86/86 [==============================] - 6s 74ms/step - loss: 0.8911 - accuracy: 0.6778
- val_loss: 0.8752 - val_accuracy: 0.6788
Epoch 12/100
86/86 [==============================] - 7s 76ms/step - loss: 0.7867 - accuracy: 0.7047
- val_loss: 0.8742 - val_accuracy: 0.6861
Epoch 13/100
86/86 [==============================] - 6s 75ms/step - loss: 0.7714 - accuracy: 0.7025
- val_loss: 0.8677 - val_accuracy: 0.6788
Epoch 14/100
86/86 [==============================] - 6s 75ms/step - loss: 0.6915 - accuracy: 0.7389
- val_loss: 0.6794 - val_accuracy: 0.7409
Epoch 15/100
86/86 [==============================] - 6s 73ms/step - loss: 0.7032 - accuracy: 0.7447
- val_loss: 0.7263 - val_accuracy: 0.7372
Epoch 16/100
86/86 [==============================] - 6s 73ms/step - loss: 0.6767 - accuracy: 0.7484
- val_loss: 1.1871 - val_accuracy: 0.6168
Epoch 17/100
86/86 [==============================] - 6s 73ms/step - loss: 0.6372 - accuracy: 0.7709
- val_loss: 0.6991 - val_accuracy: 0.7482
Epoch 18/100
86/86 [==============================] - 6s 75ms/step - loss: 0.6498 - accuracy: 0.7629
- val_loss: 0.6339 - val_accuracy: 0.7847
Epoch 19/100
86/86 [==============================] - 6s 73ms/step - loss: 0.5936 - accuracy: 0.7738
- val_loss: 0.7988 - val_accuracy: 0.7007
Epoch 20/100
86/86 [==============================] - 6s 75ms/step - loss: 0.5568 - accuracy: 0.7898
- val_loss: 0.5944 - val_accuracy: 0.7956
Epoch 21/100
86/86 [==============================] - 6s 73ms/step - loss: 0.5320 - accuracy: 0.8145
- val_loss: 0.6752 - val_accuracy: 0.7153
Epoch 22/100
86/86 [==============================] - 6s 73ms/step - loss: 0.4646 - accuracy: 0.8240
- val_loss: 0.7197 - val_accuracy: 0.7299
Epoch 23/100
86/86 [==============================] - 6s 73ms/step - loss: 0.4876 - accuracy: 0.8255
- val_loss: 0.7073 - val_accuracy: 0.7409
Epoch 24/100
86/86 [==============================] - 6s 75ms/step - loss: 0.4874 - accuracy: 0.8313
- val_loss: 0.5076 - val_accuracy: 0.8066
Epoch 25/100
```

```
86/86 [==============================] - 6s 74ms/step - loss: 0.4377 - accuracy: 0.8393
- val_loss: 1.1334 - val_accuracy: 0.6277
Epoch 26/100
86/86 [==============================] - 6s 74ms/step - loss: 0.4847 - accuracy: 0.8400
- val_loss: 0.8191 - val_accuracy: 0.6934
Epoch 27/100
86/86 [==============================] - 6s 74ms/step - loss: 0.4877 - accuracy: 0.8175
- val_loss: 0.7374 - val_accuracy: 0.7117
Epoch 28/100
86/86 [==============================] - 6s 75ms/step - loss: 0.3976 - accuracy: 0.8625
- val_loss: 0.3957 - val_accuracy: 0.8358
Epoch 29/100
86/86 [==============================] - 7s 78ms/step - loss: 0.3684 - accuracy: 0.8713
- val_loss: 0.4428 - val_accuracy: 0.8613
Epoch 30/100
86/86 [==============================] - 6s 74ms/step - loss: 0.3429 - accuracy: 0.8655
- val_loss: 0.5663 - val_accuracy: 0.7810
Epoch 31/100
86/86 [==============================] - 6s 74ms/step - loss: 0.3477 - accuracy: 0.8720
- val_loss: 0.9669 - val_accuracy: 0.6569
Epoch 32/100
86/86 [==============================] - 6s 74ms/step - loss: 0.4256 - accuracy: 0.8495
- val_loss: 0.5377 - val_accuracy: 0.8066
Epoch 33/100
86/86 [==============================] - 6s 75ms/step - loss: 0.3977 - accuracy: 0.8567
- val_loss: 0.4001 - val_accuracy: 0.8504
```

In [ ]:
```
score = model.evaluate(X_test,Y_test, batch_size=16)
```

```
26/26 [==============================] - 1s 20ms/step - loss: 0.3534 - accuracy: 0.8983
```

The model accuracy for test dataset is 89.83%.

ResNet50

When building the last layers of ResNet50, I first added the GlobalAveragePooling2D() to create feature map for each cagetory. I then added a dense layer. I tried different unit values and different activation functions and found that unit = 1500 and sigmoid activation improves the model performance the best. I also tried adding another dense layer but it didn't help. I tried several drop out values and found 0.4 the best. After tuning the last layers, I unfreeze the base model and retrain the whole model with a very low learning rate. I've tried some different values of learning rate and found lr = le-5 the best. When fit the model, I used EarlyStopping function in keras to find the optimal epoch value (=27) to avoid the issue of overfiffting.

In [ ]:
```
#Load the Xception pre-trained model
#include_top=False means that you’re not interested in the last layer of the model. You
base_model = keras.applications.ResNet50(
    weights='imagenet',
    input_shape=(img_height, img_width, 3),
    include_top=False)
```

In [ ]:
```
#To prevent the base model being retrained
base_model.trainable = False

inputs = keras.Input(shape=(img_height, img_width, 3))

# Preprocess inputs as expected by ResNet
x = tf.keras.applications.resnet.preprocess_input(inputs)
```

In [ ]:
```
#Build the last layers
#Use the functional API method in Keras to illustrate this approach
x = base_model(x, training=False)
x = keras.layers.GlobalAveragePooling2D()(x)
x = keras.layers.Dense(1500, activation="sigmoid")(x)
x = keras.layers.Dropout(0.4)(x)
outputs = keras.layers.Dense(10)(x)
model = keras.Model(inputs, outputs)
```

In [ ]:
```
model.summary()
```

```
Model: "model_19"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 input_23 (InputLayer)       [(None, 100, 100, 3)]     0

 tf.__operators__.getitem_19  (None, 100, 100, 3)      0
  (SlicingOpLambda)

 tf.nn.bias_add_19 (TFOpLamb  (None, 100, 100, 3)      0
 da)

 resnet50 (Functional)       (None, 4, 4, 2048)        23587712

 global_average_pooling2d_19  (None, 2048)             0
  (GlobalAveragePooling2D)

 dense_37 (Dense)            (None, 1500)              3073500

 dropout_19 (Dropout)        (None, 1500)              0

 dense_38 (Dense)            (None, 10)                15010

=================================================================
Total params: 26,676,222
Trainable params: 3,088,510
Non-trainable params: 23,587,712
_________________________________________________________________
```

In [ ]:
```
model.compile(optimizer='adam',
              loss=tf.keras.losses.CategoricalCrossentropy(from_logits=True),
              metrics=['accuracy'])
model.fit(X, Y, epochs=3, validation_data=(X_test,Y_test))
```

```
Epoch 1/3
52/52 [==============================] - 6s 58ms/step - loss: 2.6627 - accuracy: 0.1019
- val_loss: 2.3031 - val_accuracy: 0.1332
Epoch 2/3
52/52 [==============================] - 2s 40ms/step - loss: 2.4126 - accuracy: 0.1358
- val_loss: 2.2197 - val_accuracy: 0.1985
Epoch 3/3
52/52 [==============================] - 2s 44ms/step - loss: 2.2966 - accuracy: 0.1625
- val_loss: 2.1997 - val_accuracy: 0.1743
```

Out[ ]: <keras.callbacks.History at 0x7fc5fdb0a400>

In [ ]:
```python
# fine-tuning
base_model.trainable = True
model.summary()

model.compile(
    optimizer=keras.optimizers.Adam(1e-5),  # Low learning rate
    loss=keras.losses.CategoricalCrossentropy(from_logits=True),
    metrics=['accuracy']
)
```

```
Model: "model_19"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 input_23 (InputLayer)       [(None, 100, 100, 3)]     0

 tf.__operators__.getitem_19  (None, 100, 100, 3)      0
 (SlicingOpLambda)

 tf.nn.bias_add_19 (TFOpLamb  (None, 100, 100, 3)      0
 da)

 resnet50 (Functional)       (None, 4, 4, 2048)        23587712

 global_average_pooling2d_19  (None, 2048)             0
 (GlobalAveragePooling2D)

 dense_37 (Dense)            (None, 1500)              3073500

 dropout_19 (Dropout)        (None, 1500)              0

 dense_38 (Dense)            (None, 10)                15010

=================================================================
Total params: 26,676,222
Trainable params: 26,623,102
Non-trainable params: 53,120
_________________________________________________________________
```

In [ ]:
```python
from keras import callbacks
earlystopping = callbacks.EarlyStopping(monitor ="val_loss",
                                        mode ="min", patience = 5,
                                        restore_best_weights = True)

history = model.fit(partial, partial_labels, batch_size = 16,
                    epochs = 100, validation_data =(val, val_labels),
                    callbacks =[earlystopping])
```

```
Epoch 1/100
86/86 [==============================] - 13s 89ms/step - loss: 2.2641 - accuracy: 0.1702
- val_loss: 1.9824 - val_accuracy: 0.3869
Epoch 2/100
86/86 [==============================] - 6s 73ms/step - loss: 1.8381 - accuracy: 0.3185
- val_loss: 1.5089 - val_accuracy: 0.4672
Epoch 3/100
86/86 [==============================] - 6s 72ms/step - loss: 1.3676 - accuracy: 0.4815
- val_loss: 1.4244 - val_accuracy: 0.4307
Epoch 4/100
86/86 [==============================] - 7s 76ms/step - loss: 1.1295 - accuracy: 0.5615
- val_loss: 1.1015 - val_accuracy: 0.5839
Epoch 5/100
```

```
86/86 [==============================] - 6s 73ms/step - loss: 0.8810 - accuracy: 0.6618
- val_loss: 0.7937 - val_accuracy: 0.7263
Epoch 6/100
86/86 [==============================] - 6s 72ms/step - loss: 0.8137 - accuracy: 0.6851
- val_loss: 1.0521 - val_accuracy: 0.6314
Epoch 7/100
86/86 [==============================] - 6s 74ms/step - loss: 0.7253 - accuracy: 0.7265
- val_loss: 0.6256 - val_accuracy: 0.8066
Epoch 8/100
86/86 [==============================] - 6s 75ms/step - loss: 0.5901 - accuracy: 0.7927
- val_loss: 0.7293 - val_accuracy: 0.7628
Epoch 9/100
86/86 [==============================] - 6s 72ms/step - loss: 0.5708 - accuracy: 0.8015
- val_loss: 0.6339 - val_accuracy: 0.7664
Epoch 10/100
86/86 [==============================] - 6s 71ms/step - loss: 0.4804 - accuracy: 0.8167
- val_loss: 0.6785 - val_accuracy: 0.7737
Epoch 11/100
86/86 [==============================] - 6s 71ms/step - loss: 0.4195 - accuracy: 0.8480
- val_loss: 0.6480 - val_accuracy: 0.7774
Epoch 12/100
86/86 [==============================] - 6s 73ms/step - loss: 0.3681 - accuracy: 0.8655
- val_loss: 0.4904 - val_accuracy: 0.8467
Epoch 13/100
86/86 [==============================] - 6s 72ms/step - loss: 0.4440 - accuracy: 0.8349
- val_loss: 0.4252 - val_accuracy: 0.8467
Epoch 14/100
86/86 [==============================] - 6s 72ms/step - loss: 0.3258 - accuracy: 0.8887
- val_loss: 0.3766 - val_accuracy: 0.8796
Epoch 15/100
86/86 [==============================] - 6s 71ms/step - loss: 0.2876 - accuracy: 0.8938
- val_loss: 0.5750 - val_accuracy: 0.8066
Epoch 16/100
86/86 [==============================] - 6s 71ms/step - loss: 0.2602 - accuracy: 0.9047
- val_loss: 0.6998 - val_accuracy: 0.7664
Epoch 17/100
86/86 [==============================] - 6s 72ms/step - loss: 0.3394 - accuracy: 0.8807
- val_loss: 0.3426 - val_accuracy: 0.8686
Epoch 18/100
86/86 [==============================] - 6s 73ms/step - loss: 0.2800 - accuracy: 0.8967
- val_loss: 0.3222 - val_accuracy: 0.8978
Epoch 19/100
86/86 [==============================] - 6s 71ms/step - loss: 0.2023 - accuracy: 0.9295
- val_loss: 0.3296 - val_accuracy: 0.8686
Epoch 20/100
86/86 [==============================] - 6s 75ms/step - loss: 0.2439 - accuracy: 0.9193
- val_loss: 0.3507 - val_accuracy: 0.8540
Epoch 21/100
86/86 [==============================] - 6s 71ms/step - loss: 0.2259 - accuracy: 0.9244
- val_loss: 0.4198 - val_accuracy: 0.8504
Epoch 22/100
86/86 [==============================] - 6s 72ms/step - loss: 0.1532 - accuracy: 0.9455
- val_loss: 0.2568 - val_accuracy: 0.9015
Epoch 23/100
86/86 [==============================] - 6s 71ms/step - loss: 0.2824 - accuracy: 0.8975
- val_loss: 0.4117 - val_accuracy: 0.8431
Epoch 24/100
86/86 [==============================] - 6s 73ms/step - loss: 0.1467 - accuracy: 0.9469
- val_loss: 0.3129 - val_accuracy: 0.8796
Epoch 25/100
86/86 [==============================] - 6s 71ms/step - loss: 0.1547 - accuracy: 0.9462
- val_loss: 0.9101 - val_accuracy: 0.7080
Epoch 26/100
86/86 [==============================] - 6s 72ms/step - loss: 0.1933 - accuracy: 0.9302
- val_loss: 0.3889 - val_accuracy: 0.8796
```

```
Epoch 27/100
86/86 [==============================] - 6s 74ms/step - loss: 0.2016 - accuracy: 0.9375
- val_loss: 0.3725 - val_accuracy: 0.8577
```

In [ ]:
```
score = model.evaluate(X_test,Y_test, batch_size=16)
```

```
26/26 [==============================] - 1s 23ms/step - loss: 0.2531 - accuracy: 0.9225
```

The model accuracy for test dataset is 92.25%.

Inceptionv3

When building the last layers of Inceptionv3, I first added the GlobalAveragePooling2D() to create feature map for each cagetory. I then added a dense layer of 1200 units with relu activation and found model performance improved. I tried other activation methods and model didn't improve. I also tried adding another dense layer and performance decreased. I tried several drop out values and found 0.3 the best. After tuning the last layers, I unfreeze the base model and retrain the whole model with a very low learning rate. I've tried some different values of learning rate and found lr = le-6 the best. When fit the model, I used EarlyStopping function in keras to find the optimal epoch value to avoid the issue of overfiffting.

In [ ]:
```
seed(123)
tf.random.set_seed(123)
```

In [ ]:
```
#Load the Xception pre-trained model
#include_top=False means that you're not interested in the last layer of the model. You
base_model = keras.applications.InceptionV3(
    weights='imagenet',
    input_shape=(img_height, img_width, 3),
    include_top=False)
```

In [ ]:
```
#To prevent the base model being retrained
base_model.trainable = False

inputs = keras.Input(shape=(img_height, img_width, 3))

# Preprocess inputs as expected by ResNet
x = tf.keras.applications.inception_v3.preprocess_input(inputs)
```

In [ ]:
```
#Build the last layers
#Use the functional API method in Keras to illustrate this approach
x = base_model(x, training=False)
x = keras.layers.GlobalAveragePooling2D()(x)
x = keras.layers.Dense(1200, activation="relu")(x)
x = keras.layers.Dropout(0.3)(x)
outputs = keras.layers.Dense(10)(x)
model = keras.Model(inputs, outputs)
```

In [ ]:
```
model.compile(optimizer='adam',
              loss=tf.keras.losses.CategoricalCrossentropy(from_logits=True),
              metrics=['accuracy'])
model.fit(X, Y, epochs=3, validation_data=(X_test,Y_test))
score = model.evaluate(X_test,Y_test, batch_size=16)
```

```
Epoch 1/3
52/52 [==============================] - 7s 51ms/step - loss: 1.8960 - accuracy: 0.3505
```

```
- val_loss: 1.3733 - val_accuracy: 0.4843
Epoch 2/3
52/52 [==============================] - 1s 26ms/step - loss: 1.2632 - accuracy: 0.5561
- val_loss: 1.0339 - val_accuracy: 0.6852
Epoch 3/3
52/52 [==============================] - 1s 26ms/step - loss: 1.0522 - accuracy: 0.6295
- val_loss: 0.9495 - val_accuracy: 0.6925
26/26 [==============================] - 0s 17ms/step - loss: 0.9495 - accuracy: 0.6925
```

In [ ]:

```
base_model.trainable = True
model.summary()

model.compile(
    optimizer=keras.optimizers.Adam(1e-5),  # Low learning rate
    loss=keras.losses.CategoricalCrossentropy(from_logits=True),
    metrics=['accuracy']
)
```

```
Model: "model_12"
_________________________________________________________________
 Layer (type)                Output Shape              Param #   
=================================================================
 input_26 (InputLayer)       [(None, 100, 100, 3)]     0         
                                                                 
 tf.math.truediv_12 (TFOpLam  (None, 100, 100, 3)      0         
 bda)                                                            
                                                                 
 tf.math.subtract_12 (TFOpLa  (None, 100, 100, 3)      0         
 mbda)                                                           
                                                                 
 inception_v3 (Functional)   (None, 1, 1, 2048)        21802784  
                                                                 
 global_average_pooling2d_12  (None, 2048)             0         
  (GlobalAveragePooling2D)                                       
                                                                 
 dense_25 (Dense)            (None, 1200)              2458800   
                                                                 
 dropout_12 (Dropout)        (None, 1200)              0         
                                                                 
 dense_26 (Dense)            (None, 10)                12010     
                                                                 
=================================================================
Total params: 24,273,594
Trainable params: 24,239,162
Non-trainable params: 34,432
_________________________________________________________________
```

In [ ]:

```
from keras import callbacks
earlystopping = callbacks.EarlyStopping(monitor ="val_loss",
                                        mode ="min", patience = 5,
                                        restore_best_weights = True)

history = model.fit(partial, partial_labels, batch_size = 16,
                    epochs = 100, validation_data =(val, val_labels),
                    callbacks =[earlystopping])
```

```
Epoch 1/100
52/52 [==============================] - 11s 101ms/step - loss: 2.0680 - accuracy: 0.297
0 - val_loss: 1.4040 - val_accuracy: 0.5267
Epoch 2/100
52/52 [==============================] - 3s 61ms/step - loss: 1.4204 - accuracy: 0.4691
- val_loss: 1.5199 - val_accuracy: 0.4041
Epoch 3/100
52/52 [==============================] - 3s 64ms/step - loss: 1.2795 - accuracy: 0.4824
```

```
- val_loss: 1.1156 - val_accuracy: 0.5692
Epoch 4/100
52/52 [==============================] - 3s 64ms/step - loss: 1.1120 - accuracy: 0.5673
- val_loss: 1.0592 - val_accuracy: 0.6129
Epoch 5/100
52/52 [==============================] - 3s 64ms/step - loss: 0.8865 - accuracy: 0.6618
- val_loss: 0.9533 - val_accuracy: 0.5934
Epoch 6/100
52/52 [==============================] - 3s 64ms/step - loss: 1.1001 - accuracy: 0.5976
- val_loss: 0.7834 - val_accuracy: 0.7403
Epoch 7/100
52/52 [==============================] - 3s 64ms/step - loss: 0.7892 - accuracy: 0.7139
- val_loss: 0.6755 - val_accuracy: 0.7367
Epoch 8/100
52/52 [==============================] - 4s 72ms/step - loss: 0.6774 - accuracy: 0.7394
- val_loss: 0.8975 - val_accuracy: 0.6917
Epoch 9/100
52/52 [==============================] - 4s 71ms/step - loss: 0.6128 - accuracy: 0.7782
- val_loss: 0.6495 - val_accuracy: 0.7536
Epoch 10/100
52/52 [==============================] - 4s 78ms/step - loss: 0.6842 - accuracy: 0.7515
- val_loss: 0.5993 - val_accuracy: 0.7803
Epoch 11/100
52/52 [==============================] - 4s 75ms/step - loss: 0.5318 - accuracy: 0.8194
- val_loss: 0.5760 - val_accuracy: 0.7864
Epoch 12/100
52/52 [==============================] - 3s 65ms/step - loss: 0.3998 - accuracy: 0.8739
- val_loss: 0.4600 - val_accuracy: 0.8374
Epoch 13/100
52/52 [==============================] - 4s 72ms/step - loss: 0.4576 - accuracy: 0.8509
- val_loss: 0.7318 - val_accuracy: 0.7342
Epoch 14/100
52/52 [==============================] - 3s 65ms/step - loss: 0.7270 - accuracy: 0.7406
- val_loss: 0.4524 - val_accuracy: 0.8556
Epoch 15/100
52/52 [==============================] - 4s 72ms/step - loss: 0.4532 - accuracy: 0.8424
- val_loss: 0.6309 - val_accuracy: 0.7791
Epoch 16/100
52/52 [==============================] - 4s 72ms/step - loss: 0.4121 - accuracy: 0.8473
- val_loss: 0.5096 - val_accuracy: 0.8228
Epoch 17/100
52/52 [==============================] - 3s 64ms/step - loss: 0.4315 - accuracy: 0.8364
- val_loss: 0.3523 - val_accuracy: 0.8774
Epoch 18/100
52/52 [==============================] - 3s 64ms/step - loss: 0.2602 - accuracy: 0.9079
- val_loss: 0.3145 - val_accuracy: 0.8883
Epoch 19/100
52/52 [==============================] - 3s 62ms/step - loss: 0.2696 - accuracy: 0.8958
- val_loss: 0.3883 - val_accuracy: 0.8726
Epoch 20/100
52/52 [==============================] - 3s 61ms/step - loss: 0.2833 - accuracy: 0.9030
- val_loss: 0.3617 - val_accuracy: 0.8750
Epoch 21/100
52/52 [==============================] - 3s 61ms/step - loss: 0.3654 - accuracy: 0.8715
- val_loss: 0.3713 - val_accuracy: 0.8629
Epoch 22/100
52/52 [==============================] - 3s 62ms/step - loss: 0.3447 - accuracy: 0.8836
- val_loss: 0.3816 - val_accuracy: 0.8617
Epoch 23/100
52/52 [==============================] - 3s 64ms/step - loss: 0.3965 - accuracy: 0.8545
- val_loss: 0.3652 - val_accuracy: 0.8726
```

In [ ]:

```
score = model.evaluate(X_test,Y_test, batch_size=16)
```

```
26/26 [==============================] - 1s 21ms/step - loss: 0.3001 - accuracy: 0.9007
```

The Inceptionv3 model has a accuracy of 90.07%.

1. Model Comparison

Model	**Accuracy**
VGG-16 - University of Oxford	98.73%
ResNet -50 - Microsoft	92.25%
InceptionV3 - Google	90.07%
X-ception - Google	89.83%
EfficientNetB0 - Google	83.99%
CNN	80.34%
Fully Connected Structure	71.91%